2019 IEEE International Conference on Cloud Engineering (IC2E)

ShadeNF: Testing Online Network Functions

[1]Hui Lu, [2]Abhinav Srivastava, [1]Yu Sun
[1]*State University of New York (SUNY) at Binghamton,* [2]*Frame.io*
huilu@binghamton.edu, abhinav@frame.io, ysun59@binghamton.edu

***Abstract*—The correct implementation of network policies for "in-production" network functions is critical, as it determines the security, availability and performance of a production network. Usually, conducting network testing for these network functions in a *live* production environment is attractive, as the production environment captures the most *exact, realistic* dynamic state and vulnerabilities of the system under test. However, doing so also brings potential risks of impacting or even damaging the production system. To address this tension, we present *ShadeNF*, a novel online platform for testing in-cloud network functions in a *production-like* environment, without disrupting the real production system. ShadeNF enables such a production-like environment with an exact live clone of production network functions and real production traffic as the test traffic. In designing and implementing ShadeNF, we address several key challenges and contribute new techniques in supporting such a testing platform, including an SDN-based live, consistent snapshot approach, a new programmable forwarding plane, and a scaled test traffic generator. We implement a ShadeNF prototype upon OpenStack and demonstrate that ShadeNF successfully captures the dynamics of production systems, and effectively localizes a range of policy violations in SDN/NFV systems.**

***Index Terms*—software-defined networking, network function testing, live consistent cloning**

I. Introduction

The correct implementation of network policies for underlying network functions (NF) — such as routing, network address translation (NAT), virtual private network (VPN), load balancing, intrusion detection systems (IDS) and intrusion prevention systems (IPS) — is critical, as it determines the security, availability and performance of a production network [30], [31], [36], [50]. However, it is also notoriously known that making sure network policies are correctly implemented is challenging, even for the basic reachability policies [32], [38], [39], [43], [45], [49], [52], [53]. This becomes more challenging in today's cloud environments featured with software defined networking (SDN) -enabled network function virtualization (NFV) [34], [35], [46], where multiple tenants are hosted with much richer in-network services in the form of chained, virtualized network functions with dynamic, customized network policies [6], [7], [14], [47].

To address this problem, existing approaches have been proposed to model network behaviors, generate synthetic network traffic, and test intended network policies [33], [39], [40], [43], [54]. However, these solutions face a fundamental challenge in SDN-enabled NFV — lack of capturing *dynamics* of the production system. For example, virtual network functions (running in virtual machines) can be arbitrarily composed to realize service chaining on the fly; the chained network functions create more complex unpredictable network policies. Further, the on-demand cloud service model compounds this complexity with dynamic loads and varying network function requirements from various tenants sharing the same network infrastructure [1], [8], [10].

One (seemingly straightforward) solution may be to extend existing network models to capture dynamic system behaviors, and thus generate test traffic with broader coverage. However, despite the possibility of doing so, such model-based approaches will easily result in *state-space explosion*, which will take extensive time for completing a simple network testing task even for a small network. On the other hand, focusing on a subset of "intended" polices may reduce the state space [31], but could fail to catch some critical sources of violations in practice — in most cases, it is even hard (or impossible) to know the intended polices without really operating network functions in a production environment (e.g., with improvised changes in NFV configurations/policies).

Ideally, conducting network testing in a live production environment — complementary to model-based testing approaches — is attractive, as the production traffic captures the most *exact, realistic* dynamic state of the system under test that model-based testing tools cannot provide. However, doing so also brings potential risks of impacting or even damaging a live production environment, as mis-configured "inline" test network functions could wrongly manipulate network traffic — numerous network outages are actually caused by (tiny) mis-configurations of a live production system [9], [16], [21], [23]. It becomes more problematic in a multi-tenant cloud environment, as such misbehaviors could impact unrelated tenants [21].

In this paper, we present **ShadeNF**, a novel online platform for testing in-cloud SDN-enabled network functions in a *production-like* environment, without disrupting the live production system. ShadeNF enables such a production-like test environment (hereinafter referred to as shadow system) with an exact clone of the production network functions (to be tested). With this live clone, ShadeNF captures the dynamic state (as well as vulnerabilities) of the production system. Further, ShadeNF steers *real* production traffic to the shadow system as the test traffic, with which ShadeNF captures the dynamic state of the production workloads. Last, ShadeNF ensures that the testing be operated in a completely isolated environment with desired resources (e.g., CPU, memory and storage), hence not interfering with the production system. With these, ShadeNF offers the infrastructure-level support to deploy isolated, online network function testing services

978-1-7281-0218-4/19/$31.00 ©2019 IEEE
DOI 10.1109/IC2E.2019.00005

on behalf of tenants, running transparently to the tenants' production environment.

In designing and developing ShadeNF, we re-visit traditional wisdom, identify key challenges of existing techniques, and provide new solutions to address these challenges in the SDN-enabled NFV scenario. Particularly, we make three key contributions in supporting such a testing platform:

- **SDN-based Live Cloning.** To build a production-like enrionement, ShadeNF first copies the production network functions (to be tested) to a shadow system. ShadeNF introduces a new *live, consistent* snapshot approach (Section III-A) to clone chained, dependent network functions. ShadeNF leverages SDN-enabled programmable virtual switches — to which network functions are connected — to extensively buffer any packets produced during the inconsistent phase of a snapshot, and to re-send such buffered packets once the inconsistent phase ends. This approach not only preserves a consistent snapshot, but more importantly, reduces performance overhead (e.g., by TCP backoff) without any modifications to VMs software and legacy network flows.
- **Programmable Forwarding Plane.** To ensure strong isolation during the process of network function testing, ShadeNF places the shadow system in an isolated environment. However, to capture the dynamics of the production workloads, ShadeNF needs to bring the production traffic to this isolated shadow system. To realize this, ShadeNF creates a new traffic forwarding plane (Section III-B), which selectively, unidirectionally steers the production traffic from the production system to the shadow system with the help of *programmable forwarding plane*. This forwarding plane also helps to enable auto-chaining of arbitrary network functions (on demand) in the shadow system, resulting in a much more flexible testing framework.
- **Scaled Test Traffic Generator.** With pure production traffic as test traffic, ShadeNF may not explore sufficient cases that trigger all network policies under test. To explore broader test coverage, ShadeNF advances existing model-based approaches in generating synthetic test traffic, by taking patterns of real production traffic into account (Section III-C). ShadeNF populates synthetic test traffic with realistic traffic patterns (captured by ShadeNF automatically), such as the number of flows, protocols of flows, sizes of payload, and their combinations. Thus, these synthetic test traffic approximates the real production traffic and can be inserted on demand to the shadow system for network testing.

We have implemented a ShadeNF platform prototype upon OpenStack [15] (i.e., a popular open source cloud platform), with the above SDN-based live, consistent snapshot approach, and the new programmable forwarding plane. We have also developed the scaled synthetic test traffic generator, monitoring and resolution. Our evaluation in a realistic cloud test-bed shows that in the shadow system: (1) ShadeNF efficiently captures the dynamics of a production system at scale without affecting the production system; and (2) ShadeNF effectively detects a variety of policy violations.

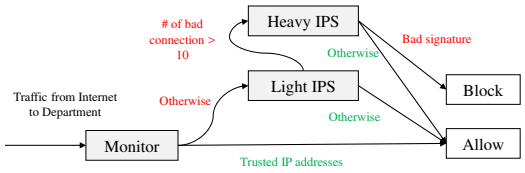


Fig. 1: An example of multi-stage policies.

Roadmap. Section II motivates our work with background and practical challenges. Section III details the design of ShadeNF, while Section IV introduces the implementation. Section V presents empirical evaluation. Section VI overviews related work and Section VII concludes the paper.

II. Motivation

A. Background

Networking function virtualization (NFV) proposes to decouple network functions, such as routing, NAT, VPNs, load balancing, and IDS/IPS, from proprietary hardware and platforms, and allows them to be virtualized and run on commodity hardware in the form of virtual machines (VM) [34], [35], [46]. NFV enables appealing usage scenarios, such as dynamic provisioning (e.g., adjusting associated resources) and elastic scaling (e.g., increasing/decreasing the number of VM instances), and is becoming a new service model in cloud [4]. Further, with one key capability being programmatically controlling network resources, software-defined networking (SDN) naturally plays an important role in NFV orchestration such as configuration of network connectivity, automation of network operations, network security and policy control [25].

In consequence, SDN-enabled NFV creates a much *dynamic* network environment in cloud, driven by on-demand service requirements from tenants [1]. For example, tenants can create customized networks with varying network topologies, functions, and policies on the fly (e.g., service function chaining [6], [7], [14], [47]). Network operators of these tenants can easily make changes to their network configurations in response to changing service requirements (e.g., to adjust the sequence of network functions, update policies, and bring in new network functions). The chained, stateful network functions enable a range of context-dependent policies that require network traffic to go through a sequence of network functions [31]. As illustrated in Figure 1, an intrusion detection system (IDS or light IPS) can reroute suspicious traffic to a deep packet inspection (DPI or heavy IPS) for further in-depth analysis — the IDS and DPI network functions create a service function chain. Looking ahead, network function chaining is poised to enable richer network processing services with more complex context-dependent policies.

B. Network Testing Challenges

The security, availability and performance of tenant networks depend closely on the correct configurations of network policies. It is particularly true for SDN-enabled NFV, where network configurations (and polices) are dynamically changing in response to on-demand network service requirements (as mentioned above). However, it is well known that making sure

network policies are correctly implemented is challenging, even for basic reachability policies (e.g., can Host A talk to Host B?) due to large state space [32], [38], [39], [43]. For instance, traditional tools may take hours to complete even for a small network setup with less than 10 nodes [31].

The system dynamics and context-dependent policies raised by chained network functions further explode the network state space, making it daunting to perform network testing. Recent effort [31] proposes promising approaches to mitigate this problem by taking in "intended" policies from the network operators to generate synthetic test traffic and inject such test traffic into the "offline" data plane. However, when applying such approaches to a live production environment, we still face the following several compelling challenges:

First, these approaches operate on a subset of offline, predefined policies, and do not directly operate in a real dynamic production environment where network policies and configurations change dynamically. In contrast, numerous network outages are caused by online mis-configurations of a live production system over operation time [9], [16], [21], [23]. Second, the synthetic test traffic does not always reflect the "on-the-wire" production traffic, as the characteristics of real production traffic are dependent on a wide variety of factors, such as running applications, traffic loads, flow types (e.g., short lived or long lived), and the combination of these factors, which is hard to capture with traffic purely generated from static models. For instance, the performance related issues of network functions are closely bound by varying live network throughput [27]; the distributed denial-of-service attack (DDoS) could only be detected by considering a large group of flows collaboratively. Last, though injecting test traffic to an "offline" data plane (by current approaches) will not disrupt the production system, it may not produce the same results as the network testing performed against the real production system directly, due to the lack of considering necessary dynamic state of the production system.

Testing network policies in a live production environment is attractive, as production traffic captures the most exact, realistic dynamic state as well as vulnerabilities of the system that the model-based testing tools cannot provide. However, testing network policies against a real production system is also restricted: It may damage and/or cause the production system to be unavailable. It becomes more problematic in a cloud environment where underlying network infrastructures is shared among multiple tenants, as network testing carried out by one tenant could affect unrelated tenants as well. For instance, one incorrect input destroyed the s3 subsystems service impacting all AWS's s3 customers in the Northern Virginia Region [21]; The failures of data plane caused by one tenant could compromise other tenants on the same data plane (e.g., in cloud, multiple VMs from different tenants share the same underlying network infrastructure) [16]; A network stress test, with high traffic loads, could impact the performance of other tenants sharing the same network infrastructure.

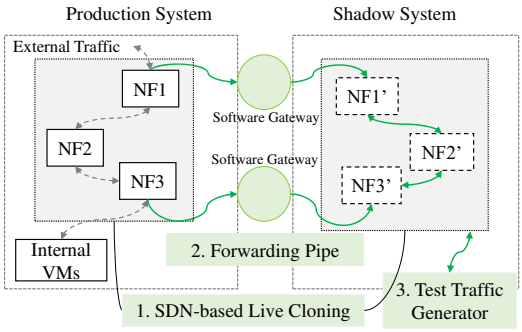


Fig. 2: ShadeNF Architecture.

III. Design

To address the challenges (in Section II-B), ShadeNF realizes online network testing for in-cloud network functions in a *production-like* environment, without disrupting the live production system. In designing and developing ShadeNF, we broadly re-visit traditional wisdom in the domain of network testing and shadow system creation. Meanwhile, we identify and address key challenges of existing techniques in the SDN-enabled NFV scenario. Figure 2 illustrates the overview of ShadeNF's architecture, consisting of three main components: The first component, SDN-based live cloning, seamlessly and consistently clones network functions (to be tested) to the shadow environment, and ensures isolation and security between the production and shadow systems. The second component, a programmable forwarding plane, selectively mirrors and redirects the live production traffic from the production system to the shadow system as the test traffic. The third component, a scaled traffic generator, further generates and injects synthetic test traffic to trigger policy-relevant behaviors that are not covered by the production traffic. We elaborate on the design of these components in the following.

A. SDN-based Live Cloning

To test network policies, ShadeNF needs to provide a *copy* (i.e., snapshot) of the network functions (to be tested) from the production system to the shadow system. The goal is to produce the same testing results in the shadow system, as performed against the production system. As a production system comprises chained, dependent network functions (i.e., in VMs) and they may complete their snapshots at different times, ShadeNF needs to further provide a *consistent copy* of these network functions — to avoid any illegal state of network functions caused by the process of shadow system creation. To this end, ShadeNF proposes a new SDN-based live, consistent snapshot approach.

A consistent snapshot ensures that the shadow system correctly captures the state of to-be-tested network functions (from the production system) at a *logical* instant of time. As shown in Figure 3, if there is no consistent snapshot, VM1 — resuming from its snapshot after time t1 — may send a packet(s), p, to its next-hop VM2, which is still in its snapshot status (note that both VM1 and VM2 reside in the production system). This causes inconsistency: VM1' — the snapshot of

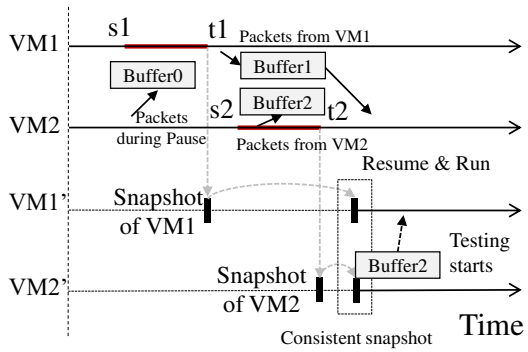


Fig. 3: Demonstration of inconsistent packets.

VM1 at time t1 — does not send p (as it has been snapshotted before VM1 sends p), while VM2' — the snapshot of VM2 at time t2 — already received p. Such inconsistency could cause *incorrect state dependency* between network functions. For example, in Figure 1, it is likely that after an inconsistent snapshot, the *state* of bad connection in the light IPS shows 9, but the heavy IPS is already triggered (supposedly by 10 bad connections). Thus, to correctly perform network testing, it is necessary to ensure a consistent snapshot.

Several distributed snapshot techniques have been proposed [28], [37], [42]. Typically, to ensure consistency, any packets sent by source VMs that have completed their snapshots (e.g., VM1) must *not* be delivered to destination VMs that have not completed their snapshots (e.g., VM2). These packets are referred to as *inconsistent packets*. One straightforward approach to deal with the inconsistent packets is to simply drop them [37] and then rely on TCP retransmission when VMs resume from snapshot. However, this approach could greatly affect transport performance (due to TCP retransmission), especially for network-intensive workloads. To mitigate this, another approach is to buffer inconsistent packets on the destination VMs side [42]. However, this approach requires marking the network header of inconsistent packets on the source VMs side, incurring non-trivial changes to the VMs' software (i.e., network protocol stack), which is less practical.

By leveraging programmability of an SDN-enabled NFV network, ShadeNF addresses this problem with a new consistent snapshot approach that minimizes performance overhead during VMs snapshot while without any modifications to VMs' software and legacy network flows. Specifically, as shown in Figure 4, each VM (being snapshotted) resides in one of two states when a snapshot starts — *in_snapshot* and *post_snapshot*. Initially, all involved VMs are in *in_snapshot*, and later transfer to *post_snapshot* after completing their snapshots. To conduct the snapshot process, ShadeNF involves two main components: an SDN controller and a snapshot controller. The SDN controller maintains state information of all involved VMs, while the snapshot controller (e.g., QEMU) executes the actual snapshot for a particular VM. At the end of a snapshot, the snapshot controller pauses a VM for a short time to save the last dirty memory, similar to traditional snapshot/migration processes. Differently, it also

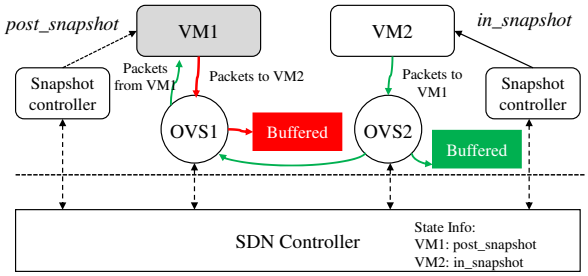


Fig. 4: ShadeNF's live, consistent snapshot.

sends a "state-change" message to notify the SDN controller to update its state (i.e., from *in_snapshot* to *post_snapshot*). After receiving the acknowledge message from the SDN controller, the snapshot controller resumes to run the VM. Figure 4 shows a concrete example where VM1 enters *post_snapshot*, while VM2 is still in *in_snapshot*.

To avoid costly TCP retransmission, inconsistent packets (e.g., packets from VM1 to VM2 in Figure 4) are buffered (instead of dropped). ShadeNF buffers the inconsistent packets on the *source* VMs side — more specifically, on the SDN-enabled virtual switches to which VMs are connected (e.g., OVS1 and OVS2 in Figure 4). In fact, once the source VM's snapshot controller sends the "state-change" message to the SDN controller (as stated above), the SDN controller not only changes the source VM's snapshot state, but also populates the VM's buffering flow rules in its virtual switch. The buffering flow rules identify all the flows that will go from the source VM to any *in_snapshot* VMs, and redirect such flows to a local buffer (e.g., Buffer1 in Figure 3). Later, when any of these destination VMs complete their snapshots, the SDN controller will be notified by their "state-change" messages; the above buffering flow rules will be cleaned up, and the buffered packets will be delivered to the destination VMs, thus reducing TCP retransmission.

Further, packets from *in_snapshot* VMs to *post_snapshot* VMs are allowed to be delivered (e.g., packets from VM2 to VM1 in Figure 4). Though, these packets do not cause snapshot inconsistency in terms of semantics of application-level message [37], they are not captured in the shadow systems. It means that when snapshot VMs resume running, this results in *packet loss* for UDP connections or *backoff* for TCP connections. To mitigate such performance overhead, ShadeNF buffers these packets as well (e.g., Buffer2 in Figure 3). As an example in Figure 4, the VM2 $\rightarrow$ VM1 packets are both delivered to VM1 and buffered on the VM2 side. The buffered traffic will be delivered as a batch to the snapshot of VM1 (i.e., VM1') in the beginning before conducting a network testing.

Last, the live cloning "pauses" source VMs in the production system for a short time during the last iteration. Though the pausing time is usually small (e.g., hundreds of milliseconds), it does bring certain performance overhead to the production system (e.g., packets may be lost during the pausing time). To mitigate the negative performance impact

to the *production* system, ShadeNF buffers packets sent to a paused VM during snapshotting (e.g., Buffer0 in Figure 3), instead of simply dropping them (like what existing approaches [37], [42] do). When the original VMs in the production system resumes running, the buffered packets will be delivered to the VMs. Thus, the production system will not observe packet loss (for UDPs) and not need retransmission (for TCPs).

To sum up, ShadeNF ensures a consistent snapshot for involved distributed network function VMs, while mitigating the performance impact by extensively buffering packets. As ShadeNF adopts a centralized SDN controller to keep track of VMs (and packets) state, *no* modifications to VMs software (e.g., network protocol stack) and network flows (e.g., adding marker packets) are required.

B. Programmable Forwarding Plane

Existing network virtualization technique provides a well-established virtualization abstraction [13], [41], where isolated virtual networks can be created for different (sub-)tenants without needing to know the underlying hardware details.

Safety. The shadow system must be isolated in the sense that the testing activities of the shadow system should not interfere with the production system. To achieve this goal, ShadeNF creates a privileged sub-tenant to host the shadow system. The cloud platform provides numerous features to allow isolation and security among multiple (sub-)tenants. ShadeNF trusts the cloud platform to implement such isolation mechanisms correctly. Further, ShadeNF only allows local cloud service providers to access the privileged sub-tenant, running transparently from the operations of production tenants.

As mentioned above, to allow the shadow system to capture the dynamics of the production workloads, it is desired to bring the production traffic to the shadow system (as test traffic), where it can be tunneled through a sequence of network functions under test. However, to ensure strong isolation, the shadow system and production system are purposely placed in separated sub-tenants, and hence cannot communicate with each other by default (i.e., they have different network namespaces and security groups). At the surface, this seems like a routing problem, however in practice, ShadeNF must overcome several constraints of cloud networking, while guaranteeing isolation and security of this traffic forwarding. To this end, ShadeNF creates a new *programmable forwarding plane* that can selectively steer the production traffic to the shadow system, while preserving the safety of the whole system. We split the design of ShadeNF's programmable forwarding plane into two parts — (1) flow identification, and (2) forwarding pipes:

Flow Identification. ShadeNF should first locate the *desired* live production flows in the production system, and then mirror and steer such flows to the shadow system (as the test traffic). ShadeNF considers three scenarios to locate the desired production flows for a particular network function under test: (a) If the network function's previous hops are part of the shadow system (i.e., the previous hops are under test), *no* production traffic will be selected. Instead, ShadeNF forwards the traffic generated by these previous hops in the shadow system to this network function, as such traffic in the shadow system reflects the latest updates of network policies. (b) If its previous hops are *not* part of the shadow system, ShadeNF needs to mirror their production flows and bring them to the shadow system. (c) If this network function is newly added (to the shadow system for testing), it could fit in either of the above two scenarios, depending on the insertion position — if it is inserted after the network functions under test, Scenario (a) applies, and vice versa.

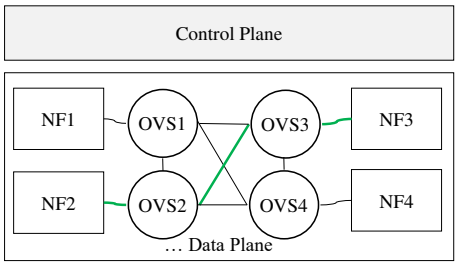


Fig. 5: ShadeNF's control plane and data plane.

In Scenario (a), ShadeNF needs to identify all the previous hops in the shadow system (given the network topology), and then to forward their traffic to the network function under test. In Scenario (b), in contrast, ShadeNF needs to bring the live production flows from the production system to the shadow system. More specifically, ShadeNF mirrors the traffic on the entry points of these previous hops: ShadeNF assumes a typical cloud setup for VMs network — every tenant VM is connected to an SDN-enabled virtual switch (e.g., Open vSwitch) [1]; inbound/outbound traffic of the VM traverses this virtual switch. Hence, this virtual switch becomes the entry point of the VM. ShadeNF mirrors the production traffic on these entry points, and redirects the mirrored traffic to the shadow system through *forwarding pipes* (introduced below).

Forwarding Pipes. ShadeNF creates *forwarding pipes* to either steer traffic through a sequence of network functions in the shadow system (i.e., in Scenario (a)), or bring the traffic from the production system to the shadow system (i.e., in Scenario (b)). A key technique that underpins forwarding pipes is *forwarding chain* [44]:

The basic unit of a forwarding chain consists of three components: (1) the network function, (2) its previous hop (i.e., a virtual switch) where network traffic comes from, and (3) its own virtual switch where network traffic enters and leaves the network function. As illustrated in Figure 5, for a flow that traverses two network functions, NF2 and NF3, the first chain {OVS_prev, OVS2_in, NF2} brings the flows to NF2, and the second chain {OVS2_out, OVS3_in, NF3} takes it to NF3. Note that, the OVS2_in and OVS2_out are the same virtual switch (i.e., OVS2) where NF2 is connected, representing inbound and outbound traffic to/from NF2 separately.

[1]The benefit of this setup is that numerous network security features to allow multi-tenancy, isolation, and security can be easily realized through this virtual switch.

OVS_prev is the previous hop of NF2. By installing flow rules in these virtual switches, ShadeNF delicately stitches all the involved forwarding units, and constructs a forwarding pipe as desired (e.g., a forwarding path from NF2 to NF3).

With forwarding chain, it is feasible to create a forwarding pipe for the above Scenario (a): ShadeNF realizes a central SDN controller that controls all virtual switches. These virtual switches as well as their associated network functions constitute the forwarding plane as illustrated in Figure 5. A specific forwarding pipe can be dynamically programmed by inserting flow rules in the related virtual switches.

However, an issue arises for Scenario (b): The previous hops of a network function (under test) reside in the production system, which are totally separated from the shadow system — strict isolation mechanisms are provided by the cloud platform between two (sub-)tenants. To establish a forwarding pipe while maintaining *safety* for the production system, ShadeNF uses a *virtual gateway* (i.e., a set of flow rules and constraints associated with an existing virtual switch) sitting at the edge of the shadow system [2]. To ensure that the production system is isolated and protected from the shadow system, the virtual gateway only allows traffic flows in the "production-to-shadow" direction, but never allows the reverse traffic (i.e., "shadow-to-production"). Hence, any traffic generated by testing activities do not affect the production system. Further, to ensure security and isolation among multiple cloud tenants, this virtual gateway is created within an individual tenant's network space (i.e., a virtual isolated network domain reserved to a cloud tenant), and thus invisible to other users.

Essentially, the virtual gateway *unidirectionally* connects the data plane of the production system and the shadow system. Hence, to create the forwarding pipe in Scenario (b), ShadeNF's SDN controller first mirrors and redirects the desired production traffic from the virtual switches of the production network functions to the virtual gateway (e.g., the first virtual switch in the shadow system); and then steers the traffic along the forwarding chains in the shadow system. Again, because of the *unidirectional connection*, no traffic flows will escape from the shadow system.

C. Scaled Test Traffic Generator

So far, ShadeNF can test the most critical behaviors of network functions triggered by live production traffic. However, with pure production traffic as test traffic, ShadeNF may not explore cases that trigger all possible network policies under test. To provide a broader test coverage, ShadeNF's scaled test traffic generator generates synthetic test traffic to test network polices not triggered by the production traffic.

ShadeNF advances traditional model-based testing approaches by considering the characteristics of real production traffic: The generation of the test traffic follows the patterns of the production traffic, hence approximating the real production traffic. More specifically, ShadeNF keeps track of the number of network flows, distribution of network protocols, protocol-specific features (e.g., status of flows), and the size of packet payloads. These network patterns are recorded by the entry point virtual switch of each network function under test, and fed to ShadeNF's traffic generator, which in turn populates the enhanced test traffic with the production traffic patterns. For example, instead of generating a single network flow given a protocol [31], ShadeNF's traffic generator creates multiple network flows in proportion to the protocol's flow number in the production traffic. Instead of adopting a fixed packet payload, the traffic generator assigns various packet sizes to different test flows based on the payload size distribution of the production traffic.

The scaled test traffic generator, running within VMs, resides in another sub-tenant, other than the production system and the shadow system for good isolation. ShadeNF uses forwarding pipes stated above to bring the generated test traffic to the network functions in the shadow system.

Test Coverage. ShadeNF intends to be a practical solution for testing most critical behaviors of network functions, triggered by live production traffic and synthetic traffic. Like most efforts in this domain [31], [48], ShadeNF is incomplete in that it is not designed to explore all possible test cases. For example, it is not designed to exhaust all possible states for an "infinite-state" system. Instead, ShadeNF allows cloud tenants to rapidly explore the most critical behaviors of network functions by capturing the dynamics of the production system and the production workloads. ShadeNF is complementary to the approaches which provide full test coverage (e.g., header space analysis [39]).

D. Test Resolution

ShadeNF's test resolution follows a model-based testing [31], [51], applied to actively test the blackbox behaviors of a system. It compares the observed behaviors of the shadow system under test (e.g., inbound and outbound network traffic of the network functions) to the expected behaviors instructed by the network policies (i.e., a traffic flow must traverse the ports mandated by the network policies).

Unlike existing approaches [31], [51] that intentionally craft a system to reach a certain known state (e.g., firewall rules, reject groups, and number of bad connections), ShadeNF uses the live production traffic, which drives a system to a *real-yet-unpredictable* state. To make such state traceable, ShadeNF uses observed traffic behaviors to retrace internal state changes of network functions (e.g., a new routing decision of one network functions indicates an internal state change, with which we can retrace its previous state, and so forth). However, we notice that, to have better test resolution, it is more desired to have network functions report internal state information automatically. In fact, a lot of internal states are logged by system software, which can be dumped through an interface. We leave this for future study.

To conduct test resolution, the entry point virtual switches of networking functions (under test) monitor both inbound and outbound traffic, and log such information (e.g., via tcpdump).

[2]In practice, ShadeNF re-uses the last-hop virtual switch of the production system, where the production flows exit the production system, to serve as the virtual gateway, thus avoiding unnecessary forwarding hops.

ShadeNF inspects the logs to check whether a traffic flow has traversed the ports instructed by the network policies (given the updated network polices and state information of involved network functions). If so, it returns with success; otherwise, a test failure (i.e., policy violation) will be reported.

E. Putting Together

ShadeNF's high-level operation policies allow tenants to request the shadow testing system in a customizable manner. The following operation information must be specified by tenants prior to using ShadeNF: (a) network functions to be tested and their connection topology (these network functions are not necessarily connected), (b) the existing/new/updated network policies of these network functions to be tested (i.e., policy specification), and (c) test coverage (e.g., using production traffic only or including synthetic traffic, and test duration). ShadeNF provides an interface for tenants to submit these information to the cloud platform, and ShadeNF, accordingly, parses the policies and deploys the testing services.

Specifically, the platform first makes a live clone of involved network functions specified by the tenants. Second, the platform starts the testing process by creating the forwarding pipes which locate, mirror and redirect the production traffic to the shadow system. While conducting testing, the entry point virtual switches log the inbound/outbound traffic, used for (i) traffic pattern characterization, and (ii) test resolution. Depending on whether tenants require synthetic traffic, ShadeNF's traffic generator may work on scaled test traffic. Last, along with the testing process, ShadeNF carries out the online test resolution to locate sources of violations.

IV. Implementation

We have implemented a prototype of ShadeNF ($\sim$3,000 LOC) on top of the OpenStack cloud platform, with its Newton release [15]. We have developed an SDN controller (using Python) as the control plane for both live cloning and forwarding plane. ShadeNF's SDN-based live snapshot approach consists of two main components: We modified the QEMU hypervisor with live snapshot support to serve as the snapshot controller (for each VM). The snapshot controller manages the lifecycle of a VM's live snapshot, while the SDN controller programs corresponding virtual switches to buffer/deliver packets. ShadeNF repurposes OVS's mirroring functions to redirect production traffic to the shadow system for the forwarding plane. For test resolution, ShadeNF employs a context-dependent traffic generator [31] to generate the *baseline* test traffic. ShadeNF further uses Scapy [17] to convert such baseline traffic to scaled test traffic with varying parameters such as sizes of packet payload, connection parallelism, protocols, and length of flows. These parameters can be configured to follow the production traffic patterns obtained from entry point virtual switches of network functions. We also implemented test resolution, which examines the observed behaviors from the shadow system with the expected behaviors extracted from network policy specification.

V. Evaluation

We have deployed and evaluated ShadeNF in a real cloud testbed with OpenStack Newton release. Our test-bed contained 16 server-grade physical machines each with two Intel Xeon quad-core processors and 32 GB memory. Each machine was installed one Gigabit Ethernet card, connecting to a Gigabit switch. In this section, we show that (1) ShadeNF's SDN-based live cloning efficiently captures the dynamics of a production system at scale with low performance overhead; (2) ShadeNF generates test traffic with broader coverage; and (3) ShadeNF effectively detects a range of policy violations.

A. SDN-based Live Cloning

To evaluate our SDN-based live, consistent cloning, we first used a "two-VM" communication case as illustrated in Figure 3. Each VM ran Ubuntu 16.04, with the same vCPU number (i.e., 2 vCPUs) but different memory sizes — VM1 with 2 GB and VM2 with 4 GB. Thus, each VM would take different amount of time to complete its live snapshot. We started the live snapshot for both VMs at the same time; VM1 would complete the snapshot ahead of VM2. We used both UDP and TCP streams, and compared *three* approaches: (1) a non-consistent snapshot, (2) a typical consistency approach [42], and (3) ShadeNF's SDN-based live cloning approach.

UDP Stream. VM1 communicated with VM2 in a bidirectional manner — VM1 sent $\sim$1000 UDP packets per second to VM2; VM2 sent $\sim$1,000 UDP packets per second to VM1. We measured lost packets and inconsistent packets (the less the better). As stated in Section III-A, lost packets occur when (1) the destination VM is paused (i.e., the last step of a live snapshot), or (2) packets are sent by an in_snapshot VM to a post_snapshot VM; inconsistent packets occur when packets are sent by a post_snapshot VM to an in_snapshot VM. Table I shows the number of lost and inconsistent packets separately for each approach.

(1) Non-consistent snapshot: A non-consistent snapshot leads to about 270 packet total losses for each UDP stream in the two-VM production system. This is caused by the snapshot "pause" time: It matches our observation that the average VM pause time is about 300 ms, while the UDP transmit rate is 1,000. In the shadow system, there is a large number of inconsistent packets for stream VM1 $\rightarrow$ VM2 (e.g., 897 in Table I, Column 3). These inconsistent packets were sent by VM1 during the time interval (t1, t2) (Figure 3), resulting in *inconsistency* between VM1 and VM2 snapshots. In addition, we observed a large number of packets losses from stream VM2 $\rightarrow$ VM1 (e.g., 1045 in Table 1, Column 3). These packets were sent by VM2 during the same time interval (t1, t2); they will never be sent by VM2's snapshot, and will not be received by VM1's snapshot, resulting in packet losses.

(2) A typical consistent snapshot [42]: Under this condition, packet losses, in the production system, only happen on stream VM2 $\rightarrow$ VM1 (e.g., 345 in Table I, Column 4), caused by the pause of VM2 during its snapshot. There are no packet losses for stream VM1 $\rightarrow$ VM2, because packets sent by VM1,

	Non-consistency	Non-consistency	Consistency [42]	Consistency [42]	ShadeNF	ShadeNF
Production/Shadow	Production	Shadow	Production	Shadow	Production	Shadow
Loss: VM1 → VM2	286	0	0	0	0	0
Loss: VM2 → VM1	267	1045	345	1238	0	0
Incon: VM1 → VM2	-	897	-	0	-	0
Incon: VM2 → VM1	-	0	-	0	-	0

TABLE I: Loss and inconsistent packets for the UDP case.

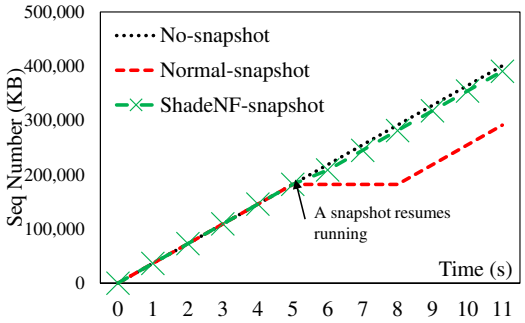


Fig. 6: TCP throughput impact.

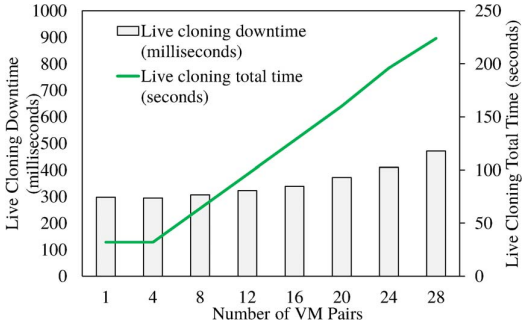


Fig. 7: Scalability of ShadeNF's live cloning.

while VM2 was paused, were buffered, and these packets were delivered to VM2 when it resumed. In the shadow system, there are no inconsistent packets observed, because, again, the inconsistent packets were buffered during the snapshot of VM2. But, we do observe a large number of packet losses on stream VM2 → VM1 (e.g., 1238 in Table I, Column 5). It is because of the same reason (as above): these packets were sent by VM2 during (t1, t2), and never sent by VM2's snapshot in the shadow system.

(3) ShadeNF's consistent snapshot: With ShadeNF's snapshot, in the production system, there are *no* packet losses for both stream VM2 → VM1 and stream VM1 → VM2 (e.g., in Table I, Column 6), because while VM1 was paused at the end of its snapshot, the packets to VM1 were buffered by ShadeNF. In the shadow system, there are *no* inconsistent packets observed, because all inconsistent packets were buffered during the snapshot of VM2. Moreover, in the shadow system, we do *not* observe packet losses from stream VM2 → VM1. It is because, again, these packets, sent by VM2 during (t1, t2), were buffered by ShadeNF and sent by VM2's snapshot in the shadow system.

TCP Stream. In the same setup as shown in Figure 3, VM1 communicated with VM2 via TCP: VM1 kept sending HTTP requests, 1,000 concurrent requests, to VM2 which ran a web server. For each HTTP request, VM2 sent data back to VM1. Figure 6 shows the impact on TCP throughput under three approaches: (1) no-snapshot, (2) a normal snapshot approach [42], and (3) ShadeNF. As in approach (2), the packets sent from VM2 to VM1 are not buffered, there is TCP backoff when the two VMs' snapshots resume in the shadow system. As shown in Figure 6, it may take 2∼3 seconds TCP backoff time, depending on how many packets were dropped during (t1, t2) (see Figure 3. In contrast, as ShadeNF buffers these packets, *no* TCP backoff is noticed in Figure 6.

Note that, ShadeNF does not guarantee avoiding packet losses completely, as there may be "on-the-wire" packets when the whole snapshot completes (in our experiments, we did not observe any packet losses). But, ShadeNF can greatly reduce packet losses by buffering packets extensively as shown above. More importantly, ShadeNF does not require any modifications to VMs' software and legacy network flows.

Scalability. We evaluate how ShadeNF's live, consistent cloning approach scales with the topology size. We use 8 physical machines for the production system, while the rest 8 physical machines for the shadow system. We run the above "two-VM" TCP case, but increase the pairs of the two communicating VMs from 1 to 28 pairs — VMs are evenly distributed among the 16 physical hosts for the production and shadow system respectively. In each case, ShadeNF conducts live snapshot for all the VMs at the same time.

As illustrated in Figure 7, as the pair number increases, the averaged downtime and total cloning time for each VM increases. More specifically, the downtime increases slightly from around 300 ms (i.e., the 1-pair case) to 470 ms (i.e., the 28-pair case). This is because, as the number of VM increases, the contention of mutiple cloning processes for the underlying physical resources increases, which slightly degrades the cloning efficiency for a single VM — each VM pauses for longer time during the last iteration. We also observe that as the pair number increases, the averaged cloning total time for each VM increases almost linearly from around 30 seconds (i.e., the 1-pair case) to 240 seconds (i.e., the 28-pair case). This is mainly because all VMs on the same host share the limited network bandwidth while cloning (e.g., 1 Gb in our setup) — as we increase the number of VMs, the network bandwidth used for a single VM decreases accordingly, resulting in longer cloning time. Notice that the total cloning time can be significantly reduced when we use higher bandwidth network such as 10/40 Gb.

Overall, when we scale up the number of VMs in the production system, the downtime (i.e., the major performance impact as VMs pause during downtime) for a single VM

increases at a reasonable rate. The total cloning time can be further reduced using a high-bandwidth network setup.

B. ShadeNF Test Traffic Generator

Next, we evaluate how ShadeNF scales when generating test traffic. We chose the context-dependent traffic generator [31] as the baseline (with policy complexity of 9), which follows a *one-packet-per-test* manner. In contrast, ShadeNF can generate test traffic with varying parameters such as payload size, flow length (i.e., in terms of packet number), and connection parallelism. As shown in Figure 8, ShadeNF brings *little* overhead in generating the test traffic. The extra overhead lies in the conversion from the baseline test traffic to scaled test traffic considering different parameters. More specifically, ShadeNF takes less than 2% extra time for a single flow with varying payload sizes (Figure 8a) and with varying packet numbers (Figure 8b). It takes less than 2x time to generate 64x connections (Figure 8c).

C. Use Cases: Finding Violations

To evaluate the effectiveness of ShadeNF in finding policy violations, we created a *production* system with OpenStack, based on a typical business-to-customer network [5], as illustrated in Figure 9a.

Specifically, the production system contains a first tier of *five* web servers (VMs), each of which has two network interfaces: a public interface and a private interface. The public interface uses Network Address Translation (NAT network function) to allow external clients to access. To reach these web servers, the external traffic (issued by *ten* client VMs, not illustrated in Figure 9a) needs to traverse a sequence of network functions including a firewall, a light IPS, and/or a heavy IPS. The private interface of the web servers uses a private address and gives certain accesses to the *ten* application servers (VMs) through a monitor (restricting web access, e.g., application server A cannot access ABC.com). To improve web performance, we use a proxy network function next to the monitor. The application servers, in turn, have at least two network interfaces: one for communication with the web servers and one for communication with the *five* database servers. The network functions sitting between application severs and database servers are the same as above: a proxy and a monitor. The key polices deployed in these network functions are highlighted in Table II Column 2.

We ran real world workloads in both web servers (Apache servers [3]) and database servers (MySQL and MongoDB). In application servers, we ran ApacheBench [2] to generate network traffic to web servers, while sysbench [22] and YCSB [24] to generate I/O traffic to database servers. We used Shorewall [18] as a NAT, a firewall and a monitor, Squid [20] as a proxy, and Snort [19] as an IPS.

At a certain point (e.g., requested by cloud tenants), we conducted a live, consistent snapshot of all network functions in Figure 9a, and created a shadow system for network testing as shown in Figure 9b: The shadow system consists of three separate network function chains. In the shadow system, we injected a variety of failures (i.e., violations of policies) in different network functions under test, as highlighted in Table II, Column 3 (some of the violations were from [31]). To locate such violations, we used both *production traffic* brought from the production system and *test traffic* generated by ShadeNF's test traffic generator. In all scenarios, ShadeNF successfully localized the failures. Table II, Column 4 specifies under which situation (e.g., production traffic or test traffic), the failures were located.

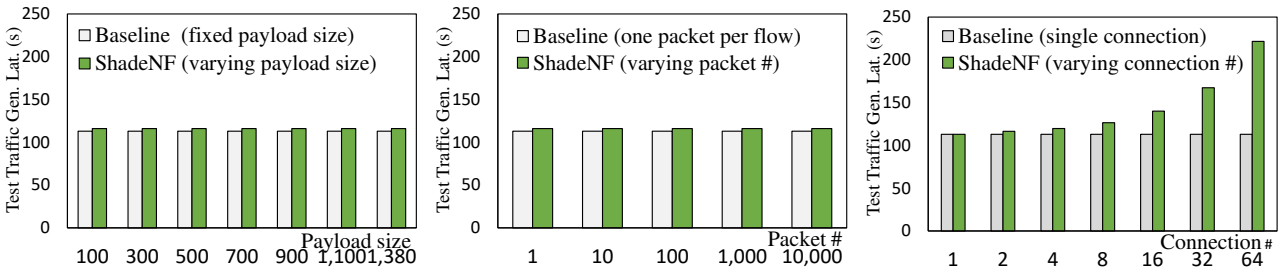
As we have observe that the production traffic can trigger all of the violations listed in Table II, demonstrating that ShadeNF serves as a practical solution for verifying critical behaviors of network functions.

VI. Related Work

A large body of literature focuses on network testing, especially on checking reachability [32], [38], [39], [43], [45], [49], [52], [53]. They check correctness in networks with stateless switches and routers, and do not capture the stateful network behaviors. Recent work starts to model complex, stateful network [30], [31], [36], [50]. The work in [36] formalized middlebox forwarding behaviors. FlowTest [30] models the entire data plane. BUZZ [31] further checks policies in stateful data planes with an expressive model considering context-dependent network policies. Symnet [50] develops models written in Haskell to capture NAT semantics. Along the same direction, ShadeNF focuses on conducting practical network testing by capturing the dynamics of live production systems using live production traffic. ShadeNF is complementary to these efforts in that it can leverage these modeling tools to generate synthetic test traffic with better test coverage such as ShadeNF's scaled test traffic generator.

To mimic an online/production system, it is common to use simulation [12], emulation [11] and shadow configuration [26]. The key difference between ShadeNF and such earlier efforts is that, instead of using static, offline configurations, ShadeNF captures the dynamic state of a complex production system, and creates a production-like test environment. The work closest to ShadeNF is POTASSIUM [42], which provides penetration testing as a service by creating an isolated system in cloud. However, our work is different from POTASSIUM in both domain characteristic (network function testing) and techniques (SDN-based live cloning and live production traffic forwarding).

There are existing efforts in consistent checkpointing [29], [37], [42]. The work in [37] used live migration to realize live snapshots by implementing a message coloring approach in routers to realize consistent snapshots. However, they do not buffer packets leading to inconsistent snapshots, but simply drop these packets. The work in [29], [42] took a snapshot using a Copy-On-Write approach via coloring and buffering packets that could lead to inconsistent snapshots. However, these approaches need to explicitly mark packet headers and thus require modifying VMs' software (i.e., TCP/IP software stack). ShadeNF advances such approaches by extensively

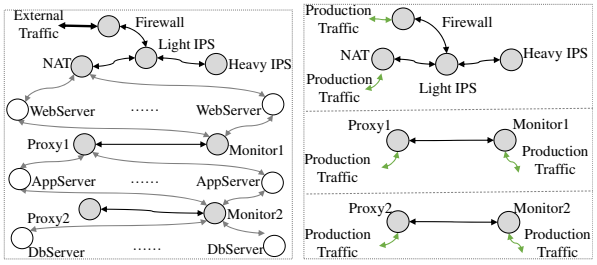


(a) Varying payload size. (b) Varying packet number. (c) Varying connection number.

Fig. 8: Test traffic generation latency: ShadeNF vs. baseline.

Network Functions	Policies	Violations	Detected by
Firewall	Allow traffic from certain IP addresses	(1) Mistakenly place client1' IP to the "reject" group (2) Mistakenly place client2' IP to the "accept" group (3) Conflicting firewall rules: Rule1, if internal connects to external IP, allow IP to access any ports; Rule 2: block any external access to internal 443 [31].	Production Traffic Production Traffic Production Traffic
NAT	Translates private addresses into public addresses	(1) Cascaded NATs using IPRewrite [31] (2) Mistakenly translate to in accessible IP addresses	Production Traffic Production Traffic
Light IPS	If Bad_conn# $>$ 10, go to Heavy IPS Otherwise, allow	(1) L-IPS miscounts by summing all the hosts [31] (2) The threshold is incorrect (3) The destination is not H-IPS	Production Traffic Production Traffic Production Traffic
Heavy IPS	Match payload signature	(1) Missing forwarding rules	Production Traffic
Monitor	Similar to Firewall	(1) Mistakenly place client1' IP to the "reject" group (2) Mistakenly place client2' IP to the "accept" group (3) Conflicting firewall rules: Rule1, if internal connects to external IP, allow IP to access any ports; Rule 2: block any external access to internal 443 [31]. (4) Application server1 cannot access web1	Production Traffic Production Traffic Production Traffic Production Traffic

TABLE II: Network policies and violations.



(a) Production system. (b) Shadow system.

Fig. 9: A "business-to-customer" network system.

buffering packets, both achieving consistent snapshot and mitigating the TCP backoff issue. Further, by taking advantage of SDN techniques, ShadeNF's live consistent snapshot approach does not need any modifications to any VMs software and legacy network flows, which is more pragmatic.

VII. Conclusion and Future Work

In this paper, we have presented ShadeNF, an online platform for testing in-cloud network functions in a production-like environment without disrupting the real production system. In designing and implementing ShadeNF, we contributed new, fundamental techniques in supporting such a testing platform including an SDN-based live, consistent snapshot approach, a programmable forwarding plane, and a scaled test traffic generator. Our evaluation results show that ShadeNF successfully captures the dynamics of the production system at scale with reduced overhead, and effectively localizes a range of policy violations under a real-world network system.

ShadeNF intends to be a practical solution for testing most critical behaviors of network functions, triggered by production workloads traffic. It trades completeness for practicality. It is incomplete in that it cannot afford exploring all possible test cases (with pure production traffic). However, this is a worthwhile trade-off, as the online testing platform will in return allow cloud users to rapidly explore the critical behaviors of network functions and perform further in-depth analysis or quick actions in response to the results. This is a desired feature in today's SDN-enable NFV environment where network policies change quickly over time and requires an online, light-weight mechanism for network testing. ShadeNF, along with the three fundamental techniques presented in this paper, serves as the building block for network function testing as a service in the multi-tenancy cloud. We believe, ShadeNF can be extended to check more complex scenarios for context-dependent network functions. We also notice that to perform more precise network testing, we need to consider application domain knowledge, which we leave for future work.

REFERENCES

[1] Amazon ec2. https://aws.amazon.com/ec2/.

[2] Apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[3] Apache http server project. https://httpd.apache.org/.

[4] Aws step functions. https://aws.amazon.com/step-functions/.

[5] Business to customer network architecture. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/ServerFarmSec_2-1/ServSecDC/2_Topolo.pdf.

[6] Dynamic service function chaining for gi-lan - intel. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/evaluating-dynamic-service-function-chaining-for-the-gilan-paper.pdf.

[7] Flexible service chaining. https://onestore.nokia.com/asset/200318/Nokia_Flexible_service_chaining_White_Paper_EN.pdf.

[8] Google cloud platform. https://cloud.google.com/.

[9] How a tiny error shut off the internet for parts of the us. https://www.wired.com/story/how-a-tiny-error-shut-off-the-internet-for-parts-of-the-us/.

[10] Microsoft azure. https://azure.microsoft.com/en-us/.

[11] Mininet: An instant virtual network on your laptop (or other pc). http://mininet.org/.

[12] ns-3. https://www.nsnam.org/.

[13] Open vswitch. http://www.openvswitch.org/.

[14] Opendaylight service function chaining usecases. https://wiki.opendaylight.org/images/8/89/Ericsson-Kumbhare_Joshi-OpenDaylight_Service_Function_Chaining.pdf.

[15] Openstack newton. https://www.openstack.org/software/newton/.

[16] Route leak causes amazon and aws outage. https://blog.thousandeyes.com/route-leak-causes-amazon-and-aws-outage/.

[17] Scapy. https://scapy.net/.

[18] shorewall. http://shorewall.org/.

[19] Snort. https://www.snort.org/.

[20] Squid: Optimising web delivery. http://www.squid-cache.org/.

[21] Summary of the amazon s3 service disruption in the northern virginia (us-east-1) region . https://aws.amazon.com/message/41926/.

[22] Sysbench. https://github.com/akopytov/sysbenchl.

[23] To err is human; to automate, divine. https://www.infosecurity-magazine.com/opinions/to-err-is-human-to-automate-divine/.

[24] Yahoo! cloud system benchmark (ycsb). https://github.com/brianfrankcooper/YCSB.

[25] Network Functions Virtualisation (NFV): Network operator perspectives on industry progress, 2014. https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf.

[26] Richard Alimi, Ye Wang, and Y Richard Yang. Shadow configuration as a network management primitive. In *ACM SIGCOMM Computer Communication Review*, 2008.

[27] Lianjie Cao, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. Nfv-vital: A framework for characterizing the performance of virtual network functions. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015.

[28] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1985.

[29] Lei Cui, Bo Li, Yangyang Zhang, and Jianxin Li. Hotsnap: A hot distributed snapshot system for virtual machine cluster. In *LISA*, 2013.

[30] Seyed K Fayaz and Vyas Sekar. Testing stateful and dynamic data planes with flowtest. In *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[31] Seyed K Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *NSDI*, 2016.

[32] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[33] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[34] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, 2014.

[35] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.

[36] Dilip Joseph and Ion Stoica. Modeling middleboxes. *IEEE network*, 2008.

[37] Ardalan Kangarlou, Patrick Eugster, and Dongyan Xu. Vnsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, 2009.

[38] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[39] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[40] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 2012.

[41] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 2015.

[42] Richard Li, Dallin Abendroth, Xing Lin, Yuankai Guo, Hyun-Wook Baek, Eric Eide, Robert Ricci, and Jacobus Van der Merwe. Potassium: penetration testing as a service. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[43] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, 2015.

[44] Hui Lu, Abhinav Srivastava, Brendan Saltaformaggio, and Dongyan Xu. Storm: Enabling tenant-defined cloud storage middle-box services. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, 2016.

[45] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*. ACM, 2011.

[46] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[47] Ahmed M Medhat, Tarik Taleb, Asma Elmangoush, Giuseppe A Carella, Stefan Covaci, and Thomas Magedanz. Service function chaining in next generation networks: State of the art and research challenges. *IEEE Communications Magazine*, 2017.

[48] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks.

[49] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[50] Matei Popovici and Costin Raiciu. Symnet: Static checking for stateful networks. In *Poster) 10th Usenix Symposium on Networked Systems Design and Implementation (NSDI 2013)*.

[51] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 2012.

[52] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On static reachability analysis of ip networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 2005.

[53] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, 2006.

[54] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.