Bring Your Own Controller: Enabling Tenant-defined SDN Apps in IaaS Clouds

Haopei Wang*, Abhinav Srivastava†, Lei Xu*, Sungmin Hong‡, Guofei Gu*
*‡SUCCESS Lab, Texas A&M University
*{haopei, xray2012, guofei}@cse.tamu.edu, ‡ghitsh@tamu.edu
†AT&T Labs - Research, abhinav@research.att.com

***Abstract*—The need of customized network functions for enterprises in Infrastructure-as-a-Service (IaaS) clouds is emerging. However, existing network functions in IaaS clouds are very limited, inflexible, and hard to control by the tenants. Recently, the introduction of Software-Defined Networking (SDN) technology brings the hope of flexible control of network flows and creation of diverse network functions. Unfortunately, enterprises lose access to the SDN controller when they move to clouds. Moreover, the cloud SDN controller is only managed by the provider administrators for security and performance reasons. To allow enterprise tenants to develop and deploy their own SDN apps in the cloud, in this paper, we introduce a new cloud usage paradigm: Bring Your Own Controller (BYOC). BYOC offers each tenant an individual SDN controller, where tenants can deploy SDN apps to manage their network. To manage these tenant SDN controllers, we propose BYOC-VISOR, a new SDN-based virtualization platform. BYOC-VISOR addresses several security and performance challenges which are specific to IaaS clouds. We show that BYOC-VISOR supports different controller platforms and diverse SDN security applications such as firewall, IDS, and access control. We implement a prototype system and the performance evaluation results show that our system has low overhead.**

I. INTRODUCTION

Software-Defined Networking (SDN) [20] brings new opportunities to control and design enterprise networks by decoupling the control plane from the data plane. It uses the logically centralized network operating system (a.k.a., SDN controller) to flexibly and dynamically manage the forwarding behaviors of the data plane. Due to these reasons, many network functions are being built and deployed as *SDN apps*. There are already many emerging SDN-based network tools/apps [14], [29] that enterprises employ to manage routine networks.

Enterprises are also embracing elastic computing offered via the cloud computing. Infrastructure-as-a-Service (IaaS) clouds (such as Amazon EC2, Microsoft Azure, OpenStack, and Google Compute Engine) provide enterprises with on-demand computing resources along with networking and storage capabilities. The pay-as-you-go model offered by the cloud computing enables enterprises to conveniently scale up and decrease resources to meet the peak demand. Cloud providers themselves employ SDN technologies to enable multi-tenancy by creating better management of tenants' networks.

While both technologies, SDN and cloud computing, provide numerous benefits to enterprises, enterprises encounter a difficult situation when they migrate to public clouds – relinquishing control over their in-house SDN controller along with the entire suite of SDN apps running atop it. The cloud provider's SDN controller that manages all OpenFlow-enabled hardware as well as software switches is not accessible to tenants. Despite tenants' demand of diverse network functions such as intrusion detection, access control, measurement, traffic engineering, and QoS, most cloud providers only offer elementary network functions such as ACL rules, load balancing, or a software suite with limited customizability. Losing access to the SDN controller deprives tenants of local and third-party SDN apps that cater their needs. Therefore, a cloud tenant desires an SDN controller to develop and deploy arbitrary SDN apps.

To this end, in this paper, we present the design and implementation details on our project called **Bring Your Own Controller (BYOC)** that provides an SDN controller, called *User Controller*, to each IaaS cloud tenant. The goal is to allow tenants to manage a network consisting of their own VMs by using the user SDN controllers onto which they can implement customized network functions (either by repurposing existing SDN apps or implementing new apps). To manage these individual SDN controllers, we propose BYOC-VISOR, a network virtualization platform which is tailored to IaaS clouds and provides customized, secure, and scalable services to tenants. Our conceptual architecture is illustrated in Figure 1. BYOC-VISOR operates from the cloud control domain and acts as a middleware layer. It provides a logical control plane instead of the actual control plane to tenants.

The design to equip each tenant with an individual SDN controller comes with several critical challenges – security, privacy, performance, and scalability – that BYOC-VISOR aims to solve. We present the main challenges and BYOC-VISOR's design to address them below:

1. Topology Abstraction: The cloud SDN controller operates on the provider's network topology to route flows dynamically to tenant networks. However, tenant SDN controllers cannot be given access to this topology as it would reveal the sensitive infrastructure-level details to tenants. Many attacks that target the cloud infrastructure (e.g., side channel attack [27]) use sniffing the physical topology and configurations as a stepping stone. Moreover, relying directly on the physical topology makes the tenants' SDN applications error-prone due to the dynamic nature of cloud systems. Some recent work (e.g., [10], [19]) proposed to translate physical topology to logical topology using loose-coupling

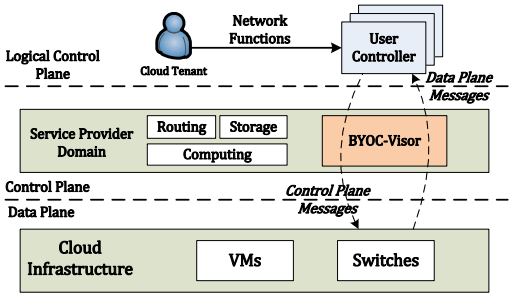

Fig. 1. BYOC-Visor Conceptual Architecture.

approaches. However, they suffer from poor performance, as discussed in Section III. We attempt to provide balanced trade-off between the flexibility and overhead. To solve this problem, we abstract the underlying topology and create the notion of a pseudo switch that is controlled by a tenant SDN controller. The abstracted topology consists of a set of tenant VMs connecting to a pseudo switch controlled by the tenant SDN controller. BYOC-VISOR's task is to map the abstracted topology into the provider's topology by programming the underlying switches. The topology abstraction scheme (***V-Topo***) prevents the leaking of sensitive provider's topology and provides the static view of the network even when the tenant VMs are frequently migrated.

2. Performance: BYOC-VISOR needs to maintain the communication between the tenant SDN controller and the cloud data plane. In particular, each data plane message should be delivered to the corresponding tenant controller (called mapping step) according to the origin of the message. Given the scale of a cloud system, if the mapping step is not efficient, BYOC-VISOR becomes the bottleneck and stalls all tenants network operations [28], [10]. This problem can also appear when a malicious tenant floods the network with the spoofed traffic from the VM to paralyze the cloud infrastructure. To solve this challenge, we design a message tagging technique called ***Message Cookie*** to improve the performance and defend against the flooding attack.

3. Security: SDN controllers influence flow routes by installing flow rules. The lack of a strong isolation among tenant SDN controllers may facilitate one tenant's flow rules to impact other tenants network traffic. In particular, malicious tenants can launch packet injection and forwarding loop attacks (described in Section IV-D2 in detail). To provide a fine-grained access control to restrict the malicious behavior from user controllers, we design a ***Message Guard*** module to monitor, profile, and filter undesired controller messages.

We incorporate all of our design choices in a prototype system of BYOC-VISOR on the GENI [5] platform. BYOC-VISOR supports multiple unmodified SDN controllers, such as Floodlight [3], OpenDaylight [7], as a user SDN controller. To demonstrate the efficacy of our system, we deployed many existing unmodified SDN-based security applications atop the user controllers. Our performance evaluation shows that BYOC-VISOR has low overhead, and scales well in clouds.

In this paper, we make the following contributions:

- We highlight the problem of migrating SDN apps to clouds, and introduce a new cloud use paradigm, Bring Your Own Controller, which provides an individual SDN controller to each tenant to design and deploy customized SDN apps.
- We describe the challenges in realizing BYOC-VISOR and present techniques– topology abstraction, message cookie, and message guard– to overcome them.
- We implement a prototype system of BYOC-VISOR, and test it with different SDN controller platforms and a variety of applications. Our evaluation results demonstrate that the system is efficient and effective.

Our paper constructs as follows. Section II provides the background knowledge and threat model. Section III introduces some related work. Section IV presents our design of BYOC-VISOR. Section V presents our prototype implementation and evaluation results. Section VI discusses the limitation and future work, and Section VII concludes the paper.

II. BACKGROUND & MOTIVATION

In this section, we provide some basic background of SDN applications, motivating examples and our adversary model.

A. SDN Application Background

SDN and its reference implementation, OpenFlow [23], bring convenient and flexible network management by separating the network control plane from the data plane. The control plane is logically centralized and works as the network operating system. In recent years, many SDN applications have been proposed, such as measurement, firewall, IDS/IPS, scanning detection, botnet detection, and DDoS detection [36], [11], [22], [13]. Besides, researchers have also proposed high-level interfaces such as FRESCO [29], Frenetic [14] to support the development of SDN applications.

In general, SDN-based network applications operate by following a series of steps sequentially as summarized here. First, the controller establishes the connection with the data plane switches. Once the connection is established, the controller or higher-level applications discover and maintain the topology information by using PacketIn and PacketOut messages with LLDP payload [17]. In the next step, the network applications obtain required network data by installing flow rules and retrieving performance counters. Finally, the SDN applications respond to network events and take relevant flow control actions (e.g. drop, set, forward) as per the enterprise policies by installing flow rules into switches.

B. Motivating Example

We allow tenants to use a user controller in the cloud similar to requesting other virtual resources such as VM, storage, and networks. Tenants employ the SDN controller to develop mainly two types of applications. In the first type, a tenant deploys an SDN application in the user controller to achieve certain network functionality. For example, a tenant needs to detect malicious scanners and redirect them to a third-party

honeypot (reflector net app in [29]). The tenant can implement the detection and redirection functions as an SDN application in a user controller. In the second type of applications, tenants deploy legacy appliances, such as middleboxes and virtual network functions, in the cloud and use their SDN controller to steer the flows towards the appliances. For example, a tenant with a Snort IDS deployed in a VM steers the network flows to the VM to monitor the traffic of other VMs. The key difference between the two types of applications is that the major processing phase occurs either in the controller or in the data plane devices.

C. Network Model

The target of our work is multi-tenant cloud networks. A multi-tenant cloud network provides individually separated cloud services to each tenant on top of a shared physical infrastructure. Our work assumes the entire physical network topology which contains both hardware and software switches is constituted in a typical Top-of-Rack or End-of-Row architecture. Each individual cloud host contains a hypervisor and multiple virtual machines (VMs). The hypervisor contains an OpenFlow-enabled software virtual switch (Open vSwitch, abbr. OVS) which connects VMs to tenant virtual networks.

D. Adversary Model

We assume that cloud providers and their physical infrastructure, including the cloud controller node and services including BYOC-VISOR running inside it, are secure and trusted. We provide each tenant with a user SDN controller and several VMs. Once a powerful adversary who takes over the VMs and/or the user SDN controller, he can launch a variety of attacks, such as flooding, spoofing, or other denial-of-service attacks on the cloud infrastructure and also affect other tenants in the cloud. The design of BYOC-VISOR, as described in Section IV, mitigates these attacks.

III. RELATED WORK

Network Virtualization: Network virtualization is a hot research topic in recent years. One work very close to BYOC-VISOR is FlowVisor [28]. While seemingly related, there are some striking differences with our work. First, FlowVisor is designed for the enterprise network under a single administrative domain, which is different from clouds that support multiple administrative domains as targeted by BYOC-VISOR. Second, FlowVisor creates parallel controllable networks by slicing the physical resources including the network topology. Since each FlowVisor slice reflects a part of the real physical topology and configurations, the slicing solution will not operate in the cloud as it does not address security & privacy concerns outlined in Section I. Finally, the peak rate of message processing in FlowVisor is about 1,200 per second [28]. This throughput does not scale well in clouds, and it will decrease with the scale of the data plane. Another system VeRTIGO [12] extends FlowVisor to allow the tenants to specify virtual links. These two slicing solutions are considered to have a tight coupling between physical and virtual topology. A different approach is based on a loose coupling between physical and virtual topology, allowing tenants to customize the virtual topologies as adopted by OpenVirteX [10] and NVP [19]. However, such solutions are too costly to be applied in clouds due to the overload of flow rules and failure to address the security threats. FlowN maps the NOX [16] API calls instead of the OpenFlow messages and uses a database instead of an in-memory complex data structure to reduce the overhead. Some network-as-a-service solutions [24] allow the tenants to specify the high-level routing policies for their traffic. However, our work provides dynamic, fine-grained and more flexible management through user controllers.

SDN Security: There are two main themes in the SDN security research. The first theme consists of systems that implement security logic in the control plane due to the controller's centralized view of the network. In this category, a series of SDN-based network security tools have been proposed [36], [11], [34], [22], [13]. The other theme addresses several security challenges in the software-defined networks itself. FortNOX [25] proposes a security enforcement kernel to address the flow tunneling attack. Avant-Guard [31] and FloodGuard [33] protect the OpenFlow control plane from the saturation attack. Rosemary [30] protects the OpenFlow control plane against malicious or faulty applications by introducing a sandbox-based solution. TopoGuard [17] addresses the network topology poisoning attack.

IV. SYSTEM DESIGN

In this section, we present BYOC-VISOR, a network virtualization platform that provides customized, secure, and scalable SDN services to cloud users. BYOC-VISOR operates as a network hypervisor and is transparent to both user controllers and the cloud data plane.

A. System Architecture

The overall architecture of BYOC-VISOR is shown in Figure 2. BYOC-VISOR consists of three main modules. The ***User Controller Hypervisor*** virtualizes standard OpenFlow interfaces for user controllers, monitors all communication messages, and blocks malicious flow rules generated by user controllers. The ***Topology Abstraction*** module achieves the V-Topo by rewriting the control plane and data plane messages. The ***Database*** module contains the profile and communication record of user controllers, the mapping table between physical and logical topology, and the message cookie information.

B. Topology Abstraction

We first describe our topology abstraction scheme.

1) Abstraction Solution: We introduce a new abstraction solution called ***V-Topo*** that provides each user controller a logical topology abstracted from the corresponding physical topology. The abstraction scheme has two steps. The first step is to decide on a physical topology representation for each tenant, and the second step is to map the physical topology to a logical topology as viewed by the user SDN controller.

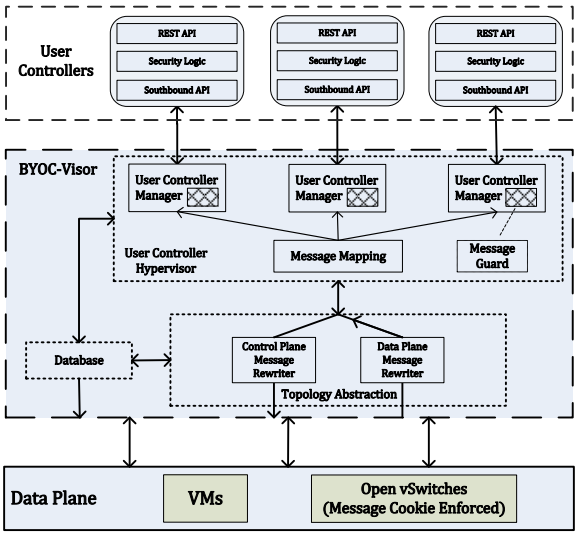

Fig. 2. BYOC-VISOR Architecture

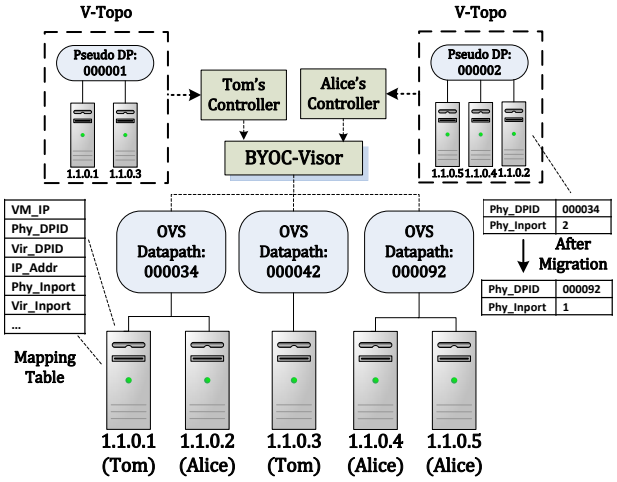

Fig. 3. Sample Abstraction

The physical topology consists of the tenant VMs and corresponding Open vSwitches (OVS) switches, running on each compute node.[1] In the logical topology, all VMs belonging to a single tenant are connected to a big pseudo switch. The pseudo switch contains virtualized configuration information (e.g., `datapath ID` and `ports`), which protects sensitive private information. Thus, each user controller has a logically separated view of the physical topology. Figure 3 shows a concrete example, where Tom and Alice are two tenants with several VMs running. Through BYOC-VISOR, Tom's user controller views one pseudo switch that is abstracted from the physical switch 000034 and 000042. Likewise, Alice's user controller views one switch abstracted from the physical switch 000034 and 000092.

In our implementation, V-Topo is achieved by modifying the message header of OpenFlow communication messages. The modification of the message header is based on the physical-logical topology mapping table maintained in the database. In the above example, Tom's data plane messages generated from the real switch 000034 is viewed as the pseudo switch 000001. Flow rules that Tom attempts to install into switch 000001 are actually installed into the real switch 000034 and/or switch 000042. In particular, if Tom's user controller installs a flow rule that steers the flow with original destination 1.1.0.1 to 1.1.0.3 then in the physical topology BYOC-VISOR installs several flow rules in both switches 000034 and 000042 and utilize the underlay cloud networking to route the flows.

Topology Abstraction module achieves the logical topology by dynamically rewriting the header fields of OpenFlow messages. For data-to-control plane messages, **Data Plane Message Rewriter** modifies the message header to insert the logical information by using the abstracted topology mapping in the database. After the modification, the data plane message rewriter sends the new messages to the User Controller Hypervisor for message mapping and distribution. The **Control Plane Message Rewriter** does the opposite work. It receives each control plane message and corresponding datapath (logical datapath) information from the User Controller Hypervisor, modifies the message header of each message by inserting the physical information, and finds the physical datapath corresponding to the logical datapath information. The control plane message rewriter sends the modified messages to the corresponding physical datapaths. Consequently, the abstraction process is transparent to both the data plane and the logical control plane.

2) Dynamic Topology Handling: Before describing the dynamic topology handling details, we first provide some background knowledge on the live VM migration. There are two types of live VM migrations: pre-copy and post-copy. A pre-copy migration copies the memory pages to the target host, suspends the original VM, and finally copies the delta memory changes changed during the process. In contrast, a post-copy migration suspends the VM first and then moves it to the target. No matter which approach is employed, BYOC-VISOR performs several actions (by changing the physical to logical topology mapping) to handle dynamism only during the "down-time" or "suspend-time". This guarantees that there should be no packet in flight during our mapping update.

BYOC-VISOR solves two challenges associated with the migration process. **Topology Consistency:** In the above example, if the VM 1.1.0.2 migrates from switch 000034 to switch 000092, the physical topology changes. However, instead of changing the whole logical topology to address the migrated resource, we only update the mapping information of this VM in the mapping table. The mapping table during the migration process is shown on the right side of Figure 3. With this approach, after the migration, the logical topology viewed by Tom still remains the same as before the migration. **Flow Rules Consistency:** A flow rule consistency ensures that the flow rules installed by the user controller should be migrated

[1]In Section IV-C2, we describe the reason for choosing only OVS as part of the physical topology.

transparently with the VM migration. During the migration state, the user controller manager migrates the corresponding flow rules and counters to the new location. If the migrated VM changes its IP address, we also verify and update the matching fields in each flow rule and counter.

C. Performance Improvement

We design ***Message Cookie*** technique to improve the scalability of BYOC-VISOR and defend against certain security threat. A message cookie has two main functions. First, it identifies the origin (from which VM) to handle spoofing threat. Second, it improves the throughput of processing mapping step.

1) Message Cookie: The throughput of the mapping step is mainly affected by the processing of PacketIn messages. There are two reasons. First, PacketIn messages are the majority traffic to the controller triggered by new data plane traffic. Second, for other messages, the processing could be easier by using a request/response pair mapping (by searching the pending request messages) to find the corresponding control logic. Existing solutions (such as FlowVisor) use a flow space mapping approach that forms an n-dimensional space based on n bits in the network packet headers. Each tenant maintains an isolated subspace that represents all packet headers belonging to the tenant. Thus, to identify the owner of a flow, we need a search algorithm to map from a high-dimensional packet header space to a tenant subspace, which is inherently slow and not suitable for large-scale cloud systems.

We introduce a novel technique, namely, ***Message Cookie***, to address the scalability challenge. Our approach is motivated by the well-known *SYN Cookie* technique. The idea is to enable switches to embed a tag to store the mapping state information within the data-to-control plane messages. We refer to the tag as ***Message Cookie***. When generating PacketIn messages, the switch can preserve mapping information into the messages by leveraging the flow table pipeline. We can utilize reserved fields such as 8 bit TOS field or unused IP header options to embed the message cookie. For example, a flow rule with "src/dst = 1.1.0.1, actions : set-tos-bits = 52, output : controller" suggests that generated PacketIn messages which satisfy the condition "src/dst = 1.1.0.1" will be marked belonging to Tom (whose UseID is 52). With this approach, it is possible to use a few flow entries to realize the mapping at each switch. Compared with the traditional flow space mapping approach, we distribute the computational workload of mapping to multiple switches instead of a single choke point. Therefore, our approach addresses the scalability challenge and achieves much higher throughput.

2) Edge-based Optimization: The overhead of the message cookie scheme depends on the location of the switches. In particular, the backbone switches, such as ToR switches or core switches, may need a large number of flow entries to implement the tagging function, making it impractical due to the limited memory space in each switch.

We propose an edge-based optimization approach to solve the problem by implementing the message cookie only inside the edge switches. An edge switch is a hypervisor software switch (Open vSwitch) that is directly connected to VMs. We note that the V-Topo's (in Section IV-B) physical topology also assumes the tenant VMs are connected to adjacent edge switches. The reasoning behind the edge-based solution in an IaaS cloud deployment is that each edge switch is normally connected to no more than 30 VMs [35]. Thus, it is possible to enforce efficient and simple tagging function at the edge switches with low overhead (a few flow entries).

D. User Controller Hypervisor

1) User Controller Manager: The main function of the User Controller Hypervisor is to virtualize interfaces for the user controllers. The user controller manager leverages the standard OpenFlow protocol to communicate with the user controller. For the Connection Initiation and Topology Discovery request messages, the manager automatically generates the data-to-control plane messages to respond to the user controllers. For example, in response to the negotiation messages (FeatureReq/Res, SetConfig), the manager provides the configuration of the abstracted topology to the user controller. For the OpenFlow messages in the attack detection and response actions stages, the manager simply relays these messages. Although the communication (we define as **Hypercalls**) between the manager and the user controller uses the OpenFlow protocol, it is not a "real" OpenFlow communication; in fact, it is between the cloud control plane and logical control plane.

2) Message Guard Module: Another major function of the User Controller Hypervisor is to restrict the behaviors of user controllers and enhance the security of our system by checking the Hypercall messages. In each User Controller Manager, there is a **Message Guard** module. This module continuously monitors the hypercall communication and detects any possible malicious behaviors from user controllers. More specifically, the message guard module introduces several security features including fine-grained access control, profiling, and rate-limiting. In case of attacks, our system quickly blocks the malicious tenant and removes all counters and flow rules that are installed by the attacker. The cloud administrator can gather detailed information on the identified malicious tenants for further fine-grained analysis.

The message guard module limits the cumulative number of both control-to-data plane messages and the rate of data-to-control plane messages for each tenant. Limiting the control-to-data plane messages is to restrict the data plane resources that one user controller can consume. On the other hand, limiting the data-to-control plane messages is to prevent the flooding attack originating from VMs.

We also provide fine-grained access control on all control messages generated by user controllers. The purpose of access control is to guarantee the control logic enforced by one tenant should not affect other tenants' network traffic. The message guard module monitors and verifies all control-to-data plane messages. We only allow two types of control-to-data plane messages, namely, FlowMod and StatsReq messages. The FlowMod message is used to insert, modify, or delete

flow rules. To verify if a tenant is allowed to perform certain operations, source or destination address in each matching rule or header fields after modification should be within the scope of the address space of the tenant. The action in each flow rule can take one of these values: DROP, CONTROLLER, SET, or FORWARD (means drop, trigger PacketIn to the controller, modify the packet header, or forward this flow). The StatsReq message is to request the traffic statistics (flow statistics, port statistics, etc.). We only allow the flow statistics requests and the matching rule of the flow should have the same requirement as for the flow rules.

However, the above access control policies are not enough to block all malicious behaviors from user controllers. It is difficult to predict the effect of the action field in each flow rule. Especially, we consider two new action-based attacks. *Packet injection* attack means the user controller can use the "modify" action to change the header fields of packets to inject arbitrary packets to the cloud network. Those spoofed packets may affect other tenants' VMs or even user controllers. *Forwarding loop* attack means the user controller can install flow rules to form a routing loop of its own traffic in the cloud. By increasing the traffic quantity, the routing loop can obstruct the cloud infrastructure.

Algorithm 1 switch_iteration

Input: r: new flow rule,
sw: switch ID in which r will install,
$s[].r\_list$: all installed flow rules of every switch

1: **if** policy_check (r) == $Malicious$ **then**
2: **return** $False$
3: **end if**
4: **if** $Action.Forward \in r.actions$ **then**
5: sw' = $sw.switchID(r.actions.forward)$
6: **if** $Action.Modify \in r.actions$ **then**
7: $r.match$ = modify ($r.match$, $r.actions.modify$)
8: **end if**
9: **for** $i \in s[sw'].r\_list$ **do**
10: $r'.match$ = $i.match \cap r.match$
11: $r'.actions$ = $i.actions$
12: **if** !$(switch\_iteration(r', sw'))$ **then**
13: **return** $False$
14: **end if**
15: **end for**
16: **return** $True$
17: **end if**

We design an iteration algorithm, shown in Algorithm 1, to achieve the above-mentioned goals. Our algorithm dynamically verifies the new flow rules in the physical topology of each tenant because the physical topology is tied to the actual behavior of the network. The input to this algorithm is a new flow rule and a set of already installed flow rules. Existing real time data plane verification tools such as VeriFlow [18] are of limited use because flow rules with "set" action change the header space of packets that cannot be handled by these tools. Our algorithm generates the derived forwarding rules from the new flow rule hop by hop. Then, we verify if the derived forwarding rule violates some access control policies. For example in Figure 3, given that Tom installs a rule that steers the flow from the original destination 1.1.0.1 to 1.1.0.3 into his pseudo switch, the flow rule that will be installed into the OVS switch 000034 is: "Sw000034 : * -> 1.1.0.1 : set(1.1.0.1 -> 1.1.0.3), forward Sw000042", given the existing flow rule in the data plane is: "Sw000042 : *-> 1.1.0.3 : set(1.1.0.3 -> 1.1.0.1), forward Sw000034". After running 2 iterations in our algorithm, the derived forwarding rule violates the forwarding loop policy since it loops back to the original switch. For messages that do not meet the above-described access control policies, the user controller manager drops the control message and returns an **OFPET_EPERM** error.

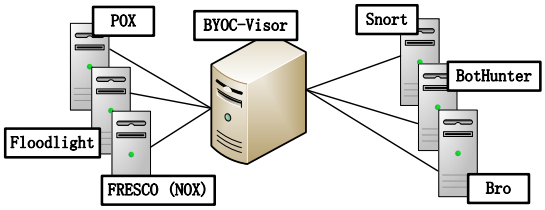

Fig. 4. Test Environment for Case Study

V. Evaluation

To demonstrate the practicability and efficiency of the BYOC-Visor design, we develop realistic SDN security applications atop various user controllers and evaluate the performance and scalability of BYOC-Visor.

A. System Implementation

We have implemented a prototype system of BYOC-Visor based on the libfluid [6] library bundle. To evaluate the dynamism handling and performance overhead, we employ resources including VMs, hypervisors, and OpenFlow-enabled switches to emulate the cloud environment on the GENI [5] platform. To evaluate the scalability, we create a testbed using three host machines with dual-core Intel Core2 3GHz CPU running 64-bit Ubuntu Linux. The first machine emulates the cloud data plane using Mininet [21], and a benchmark tool CBench [1] as the bulk messages generator, another is to run BYOC-Visor, and the third operates as user controllers. Our system currently supports OpenFlow 1.0 specification and is compatible with most of the OpenFlow controllers as user controllers. We demonstrate our system using both the OpenFlow-based apps and legacy network functions.

B. Case Study

In Section II, we discuss two types of SDN applications with the key difference being the location of the major processing phase – inside the controller or data plane devices. In our evaluation, we design and deploy three SDN-based network functions to work atop user SDN controllers, and three legacy network functions to operate inside the VM in the data plane. The test environment is shown in Figure 4.

Control-plane network functions: We develop three SDN apps on top of three different user SDN controllers. The first two are firewall SDN apps which are developed to work with POX [8] and Floodlight [4]. The third SDN app is a reflector net application developed upon FRESCO [29] that executes

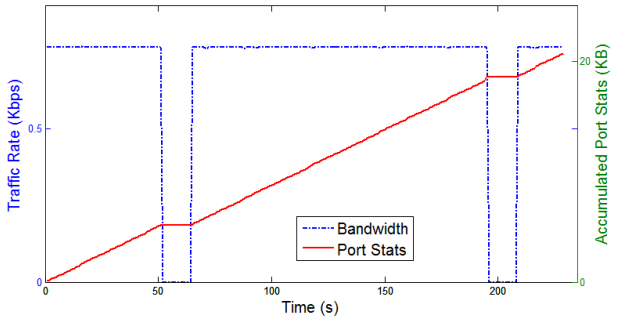

Fig. 5. Communication Bandwidth and Port Stats during Migration

on a NOX controller. We notice that BYOC-VISOR supports the correct operations of all three original SDN-based network functions on diverse OpenFlow controller platforms, without any modification on the controller or app side.

Data-plane network functions: We deploy three different legacy network applications, namely Snort [32], BotHunter [15], and Bro [9]. We install these legacy applications in three VMs as middle-boxes (a.k.a. NFV, Network Function Virtualization). On top of the user controller, we develop an SDN application using the FRESCO platform [29]. The SDN application steers the network traffic destined to tenant VMs towards the Snort/BotHunter/Bro VMs. When the middle-box VMs accept the traffic, it steers them back to the destination VM. In our testing, all scenarios work smoothly as expected. These applications demonstrate the effectiveness of our system in allowing tenants to design and deploy SDN apps. We also hope that these examples would provide guidelines for tenants to develop more such apps on BYOC-VISOR.

C. Dynamic Handling

We first test the ability of BYOC-VISOR to handle frequent topology changes. We design an experiment to verify that the logical topology observed by the user controller remains unchanged even with the frequent VM migration. We build experimental topology as shown in Figure 3 using the GENI [5] platform and use two VMs with IP addresses 1.1.0.4 and 1.1.0.5. At the beginning, two VMs are connected to the same switch. Later, one VM (1.1.0.5) migrates to switch 000042, and then migrates again to switch 000034. We generate communication traffic between the two VMs and record the traffic rate. To verify that the V-Topo remains unchanged from the user controller side, we send StatsRequest messages from the user controller to the pseudo switch in V-Topo to query the real time traffic rate at the port[2], which is initially connected to VM 1.1.0.5 and show the accumulated traffic in Figure 5. We observe that the migration occurs twice at about 51s and 195s because the bandwidth suddenly decreases to zero. During the migration, there is no traffic passing through the port. The results show that even when the VM moves to another location in physical topology, the VM is still connected to the original port of the pseudo switch. The experiment results verify that the logical topology observed by the user controller is stable and BYOC-VISOR elegantly handles VM migration.

[2]In Section IV-D2, we mention that we only allow the user controller to query the flow statistics not the port statistics. Here we temporarily relax this assumption only to conduct this experiment.

D. Performance Overhead

BYOC-VISOR inserts an additional middle layer and unavoidably adds extra overhead to the system. From the tenants' perspective, there is an additional latency while sending and receiving messages. To quantify the latency overhead, we evaluate the increased response time for the two most commonly used OpenFlow request messages– PacketIn and StatsReq/Res– with and without our system. The PacketIn message is used for the data plane to send a network packet to the control plane when a new flow arrives in or a flow entry sends a specific flow to the controller. The StatsReq message is from the controller to query the data statistic, and the data plane returns a response message with the statistics.

For the PacketIn message experiment, we set up an environment with a VM with two network interface cards attached to an OpenFlow-enabled switch in GENI. An OpenFlow application continuously sends randomly generated packets (with a rate of 100 packets per second) to the switch through one interface. The application simultaneously receives PacketIn messages from the other interface that is connected to the OpenFlow control port of the switch. Thus, this application is able to measure the response interval between sending the packet and receiving the PacketIn messages. The evaluation results are shown in Figure 6(a). We observe that without BYOC-VISOR, the average delay between each pair of packet and PacketIn is about 0.25ms. With BYOC-VISOR, the average delay increases to about 0.37ms. We note that this communication overhead is mostly added only on the first packet and gets amortized across the duration of the flow.

For the StatsReq/Res message experiment, we set up another environment with a VM as an OpenFlow controller that connects to several OpenFlow-enabled switches. An application queries the flow statistics from the switches at a peak rate supported by the hardware. The application also measures the delay between each pair of request and response. The evaluation result is shown in Figure 6(b). We notice that without BYOC-VISOR, the average delay is about 0.45ms, and with our system is 0.52ms, which is a reasonably small overhead.

E. Scalability

To evaluate the scalability of BYOC-VISOR, we create a 3-machine setup as described in Section V-A. All user controllers run a firewall app. We first determine the CPU utilization of BYOC-VISOR under normal circumstances except the migration situation. We measure the CPU utilization when using a different number of user controllers and message rates. To measure the effect of different message rates, we use one VM to continuously send packets at different rates to the OVS to trigger PacketIn messages for the user controller. To measure the effect of various numbers of user controllers,

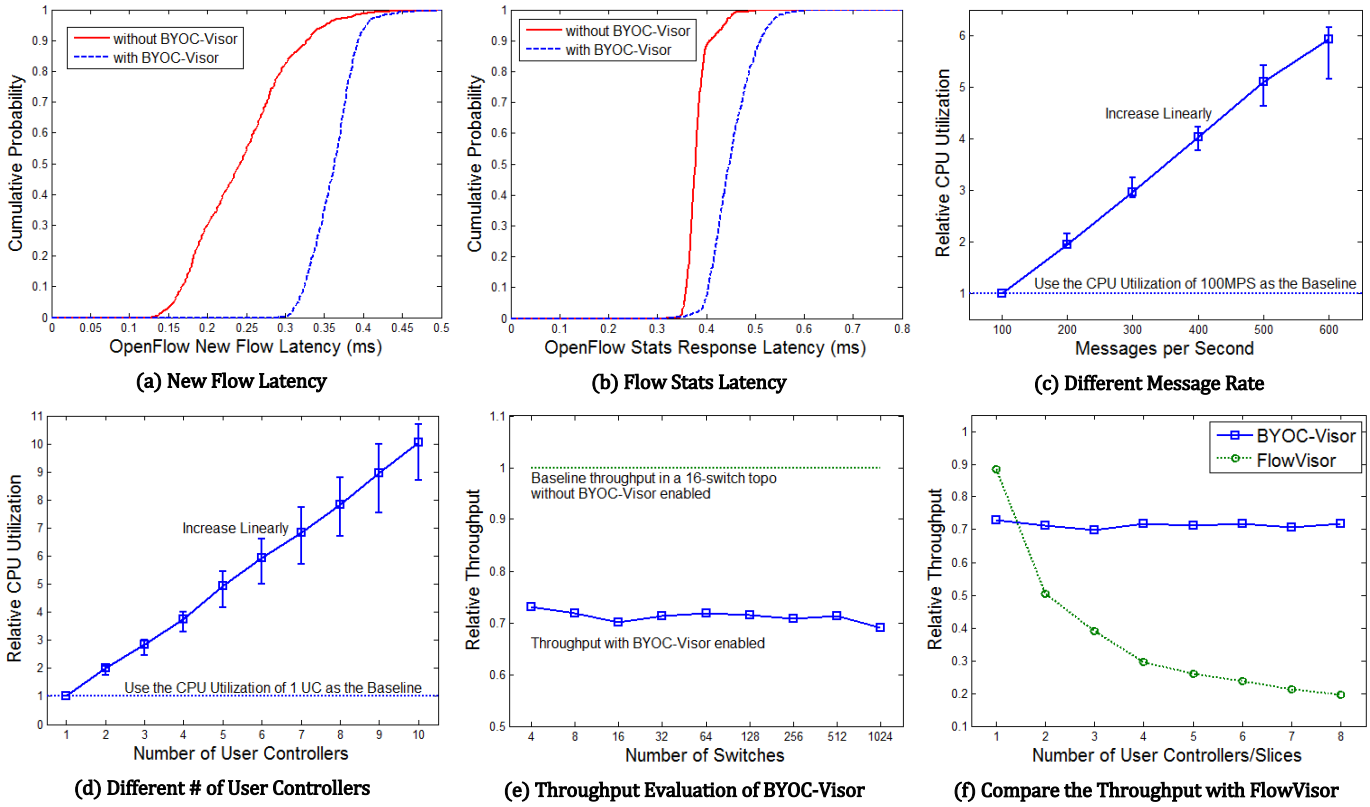

Fig. 6. Performance and Scalability Evaluation Results

BYOC-VISOR connects to several user controllers and sends messages to each user controller at a constant rate. We measure the CPU utilization at every one second for an extended period of time and calculate the average. Since different types of hardware devices may have different capabilities, it is both difficult and insignificant to compare the absolute CPU utilization among them. A better metric is to observe a *growth* in CPU utilization with an increase in message rates and the number of user controllers. Thus, in this experiment, we show the CPU utilization growth using the relative CPU utilization and compare it with a baseline value. The baseline utilization value is generated using 100 MPS (messages per second) in the first experiment, and a single controller in the second experiment.

The results are shown in Figure 6(c)(d). We observe that the CPU utilization scales linearly with the number of user controllers and message rates. This is consistent with the theoretical analysis, and is an acceptable growth trend. In practice, cloud administrators may deploy multiple instances of BYOC-VISOR to balance the load among tenants.

Secondly, we measure the message mapping throughput. One benefit of message cookie is to avoid searching the entire mapping flowspace for each PacketIn message. This suggests the throughput of the message mapping process should not be affected by the scale of the data plane topology. To validate the hypothesis, we set up a message throughput experiment, using a benchmark tool *Cbench* [1] to evaluate the throughput with different scales of topology (by increasing the number of OVS switches and VMs in the topology). In our testing topology, each OVS connects to 8 VMs, while each user controller manages 4 VMs, randomly assigned to it.

Like the CPU utilization experiment, we measure the relative growth in throughput instead of comparing the absolute values. We measure the baseline throughput using the Floodlight controller, without running BYOC-VISOR, in a 16-switch topology. We evaluate a relative throughput compared with the baseline by scaling the topology from 4-switch to 1024-switch and executing a single instance of BYOC-VISOR in a single thread. The results are shown in Figure 6(e). We observe that the throughput is not impacted with the topology scale, outperforming FlowVisor whose throughput decreases linearly under the same condition as described in [28]. The results prove that our system scales well with a large number of switches in a cloud environment. There are several studies about the OpenFlow controller performance [2], [30]. These studies measure a baseline throughput of the Floodlight controller in a dedicated server machine using the same 16-switch topology, and the value is about 100k messages per second. Using the same method, we estimate that BYOC-VISOR can process about 70k messages per second.

Finally, we design an experiment that measures the throughput performance using the different number of user controllers to directly compare with FlowVisor. We simply use one user controller corresponding to one slice in FlowVisor. In

this experiment, we use the 16-switch topology and test the message throughput of both single thread BYOC-VISOR and FlowVisor. Each user controller/slice manages the traffic from 8 VMs. We set the baseline value same as in the previous experiment and show the relative throughput in Figure 6(f). The results verify that the throughput of FlowVisor decreases with the topology scale, which is the same as the theoretical results mentioned previously [28]. However, the throughput of BYOC-VISOR does not decrease with scale. Message cookie distributes the computation workload to the edge switches, making the throughput independent of the topology scale.

VI. DISCUSSION AND FUTURE WORK

Our current implementation of BYOC-VISOR supports the OpenFlow v1.0 protocol. We plan to support the latest OpenFlow v1.5 in our future work. Also, the user controller may have the inconsistent update issue that implies all switches cannot be updated atomically. Note that this issue is within each individual user controller, and it is not the responsibility of BYOC-VISOR. User controllers can directly leverage the existing solution [26] to address the inconsistent update issue. Finally, we implement the message cookie by installing flow rules to add a tag to each message. This approach avoids any hardware-level changes, creating a flexible yet less optimal solution. Alternatively, we can improve the performance by modifying the OpenFlow switches to enforce the message cookie function in the circuit to avoid extra flow tagging rules.

VII. CONCLUSION

We aim to provide tenant-defined SDN applications in IaaS cloud networks. To this end, we offer an individual user SDN controller to each tenant. This approach requires addressing several new challenges: topology abstraction, performance, and security. We present BYOC-VISOR, a new SDN virtualization platform to provide customized and scalable SDN services to cloud users. We measure the overhead and scalability performance with a prototype implementation of BYOC-VISOR. Our evaluation results show that BYOC-VISOR scales well in the cloud and only adds minor latency overhead.

VIII. ACKNOWLEDGEMENTS

This material is based upon work supported in part by the the National Science Foundation (NSF) under Grant no. CNS-1314823 and ACI-1642129, and a Google Faculty Research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF and Google.

REFERENCES

[1] Cbench Controller Benchmarker. https://github.com/andi-bigswitch/oflops/tree/master/cbench.
[2] Controller Performance Comparisons. http://archive.openflow.org/wk/index.php/Controller_Performance_Comparisons.
[3] Floodlight Controller. http://www.projectfloodlight.org/floodlight/.
[4] Floodlight Firewall Module. https://github.com/floodlight/floodlight/tree/master/src/main/java/net/floodlightcontroller/firewall.
[5] GENI: Global Environment for Network Innovations. https://www.geni.net/.
[6] Libfluid: The ONF OpenFlow Driver.
[7] OpenDaylight Controller. http://www.opendaylight.org/.
[8] OpenFlow Firewall Application. https://github.com/hip2b2/poxstuff/blob/master/of_firewall.py.
[9] The Bro Network Security Monitor. https://www.bro.org/.
[10] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow. Openvirtex: Make your virtual sdns programmable. In *HotSDN*, 2014.
[11] J. R. Ballard, I. Rae, and A. Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. In *WREN*, 2010.
[12] R. Doriguzzi Corin, M. Gerola, R. Riggio, and F. De Pellegrini. Vertigo: Network virtualization and beyond. In *EWSDN*, 2012.
[13] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Security*, 2015.
[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.
[15] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. Detecting malware infection through ids-driven dialog correlation. In *USENIX Security*, 2007.
[16] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. In *ACM CCR*, 2008.
[17] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS*, 2015.
[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.
[19] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *NSDI, 2014*.
[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. In *ACM CCR*, April 2008.
[21] Mininet. Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/.
[22] A. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *WREN*, 2009.
[23] OpenFlow. Innovate Your Network. http://www.openflow.org.
[24] L. Peterson, S. Baker, M. D. Leenheer, A. Bavier, S. Bhatia, M. Wawrzoniak, J. Nelson, and J. Hartman. Xos: An extensible cloud operating system. In *BigSystem*, 2015.
[25] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for openflow networks. In *HotSDN*, 2012.
[26] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, 2012.
[27] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.
[28] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI*, 2010.
[29] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS*, 2013.
[30] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. In *CCS*, 2014.
[31] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *CCS*, 2013.
[32] Snort. . http://www.snort.org/.
[33] H. Wang, L. Xu, and G. Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *DSN*, 2015.
[34] Y. Wang, Y. Zhang, V. Singh, C. Lumezanu, and G. Jiang. Netfuse: Short-circuiting traffic surges in the cloud. In *ICC*, 2013.
[35] Z. Xu, H. Wang, and Z. Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security*, 2015.
[36] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *PAM*, 2013.