Identifying State Inconsistency in OpenStack

Yang Xu, Yong Liu
ECE Department, New York University
yx388@nyu.edu/yongliu@nyu.edu

Rahul Singh, Shu Tao
IBM T. J. Watson Research Center
rahulsi@us.ibm.com/shutao@us.ibm.com

ABSTRACT

In Software Defined Networks (SDN), users manage network services by abstracting high level service policies from lower level network functions. Edge-based SDN, which relies on end hosts to implement lower-level network functions, has been rapidly developed and widely adopted in cloud. A critical challenge in such an environment is to ensure that lower level network configurations, which are distributed in many end hosts, are in sync with high-level network service definitions, which are maintained in the central controller, as state inconsistency often arises in practice due to communication problems, human errors, or software bugs. In this paper, we propose an approach to systematically extracting and analyzing the network states of OpenStack from both controller and end hosts, and identifying the inconsistencies between them across multiple network layers. Through extensive experiments, we demonstrate that our system can correctly identify network state inconsistencies with little system and network overhead, therefore can be adopted in large-scale production cloud to ensure healthy operations of its network services.

1. INTRODUCTION

In Software Defined Networks (SDN), users only define high-level services and leverage programmable controller to send configurations to distributed forwarding devices to realize low-level network functions. This makes network management flexible and easy. Depending on the forwarding devices that controller configures, SDN could be categorized into two types: *core-based* and *edge-based*. In core-based SDN, the controller directly configures core network devices, such as OpenFlow switches and routers [24, 5]. In edge-based SDN, controller configures network edge devices, i.e., end hosts that act as virtual switches or routers. Recently, edge-based SDN has been rapidly adopted in the emerging cloud environments, e.g., OpenStack [25], CloudStack [9], Eucalyptus [10], etc.

A critical challenge in SDN is to ensure the consistency between high-level network service definitions and low-level configurations. This problem is more prominent in edge-based SDNs, because: (1) in such environments, low-level network configurations are distributed across potentially many end hosts, and (2) virtual switches or routers implemented in end hosts are less reliable than dedicated network devices, hence are more likely to face various types of errors during their operations. If such problem arises, misconfigurations on end hosts can potentially break the intended network functions. In a multi-tenant cloud, this may even lead to security breaches by exposing private network traffic to unauthorized users.

In this paper, we study the problem of identifying state inconsistencies between controller and end hosts in OpenStack. In OpenStack forum, many operators have reported their encountered network inconsistency problems [12, 11]. In our experience, those inconsistencies often arise due to the following reasons:

- *Communication Problems:* For network services in OpenStack, controller sends network configurations to end hosts through asynchronous messages. If messages are lost or communication is disrupted during the configuration process, the states between these two can become out of sync. Although one can use TCP to enhance reliability, message losses still happen due to various reasons, e.g., sending buffer becomes full [21], message broker dies [2], etc.
- *Human Errors*: The commodity servers used to build today's cloud are not always reliable. System admins often need to reboot, patch or repair the system manually. This process can potentially introduce errors in virtual network configurations.
- *Software Bugs*: Edge-based SDN implementations are not bug-free. We have experienced cases in which the network configurations pushed into end hosts do not exactly reflect the network policies defined at the controller. And some bugs are hard to detect before real deployment.

To address those problems, we propose an approach to systematically identify the state inconsistencies between controller and end hosts. We model network states at three layers. In our approach, we will first extract the state data from the SDN controller, which is typically stored in a database. By parsing this data, we obtain the network states that should be implemented in the network. Then, we deploy a light-weight agent on each end host to extract the virtual network configurations, and parse out the network states that are actually implemented. The two sets of network states are sent to a central verification server, which compares and identifies any inconsistencies between them.

Toward developing the approach outlined above, we made several contributions in addressing the following challenges:

- *Network State Abstraction*: in large cloud environment, large set of network configuration data needs to be processed from the edge devices. We developed succinct representations for layer 2-4 network states, respectively, and provided mechanisms to efficiently map raw configuration data into such state representations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOODSTOCK '97 El Paso, Texas USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

- *Sheer Volume of Data Processing*: Another practical challenge is to deal with the sheer volume of configuration data to be processed. In particular, for L4 state parsing, security group rules, typically implemented with iptables, need to be parsed and analyzed for each VM. For a production cloud environment that contains thousands of (or more) VMs, this can become a daunting task. We develop several methods in addressing this challenge. First, we develop an efficient method of traversing the iptable rules and use the Binary Decision Diagram (BDD) to succinctly represent and analyze these rules. Second, to speed up the parsing process, we developed smart *L4 state cache* to avoid repetitive network state parsing for VMs with the same configuration. Finally, we design a *two-level verification* method to further speed up the verification process.

- *System Design and Implementation*: We implemented a verification system for the OpenStack cloud, developed various data processing and caching mechanisms to reduce its overhead. We also conducted extensive experiments to demonstrate that our system can quickly identify network state inconsistencies in real cloud environments.

1.1 Related Work

Consistency maintenance is important for network management system [20, 27]. But in the real system, inconsistencies could still occur due to communication problems, human errors or software bugs. There have been several studies on checking or debugging SDN state problems between the control plane and the data plane. Heller et al. [15] provided a survey on the existing studies and tools for troubleshooting SDN. Specifically, [26, 13] studied high-level abstractions for easier OpenFlow configurations. The work in [18, 8, 22, 3] provided various techniques, e.g., model checking, boolean expressions, SAT solver, to detect the inconsistencies between the defined network policy and the actual network state, in a core-based SDN environment. To reduce the overhead of such checking, Khurshid et al. [19] and Kazemian et al. [17] proposed using trie structure or dependency graph to allow incremental checking. Also related are the work on debugging SDN configurations, either by passively sniffing packets [29] or by actively sending test packets to test the SDN systems [14]. All of the existing studies were focused on the state inconsistency problem in core-based SDNs, e.g., OpenFlow. They assume the inconsistencies are due to the discrepancy between admins' logical designs and their actual flow-level implementations. Since OpenStack only offers coarse-level commands to admins and don't allow them to work directly on flow-level implementations, OpenStack don't have those type of inconsistency problems, and the techniques developed for core-based SDNs cannot be directly used for OpenStack.

The study in [16] shows that the state inconsistency problem in edge-based SDN is becoming critical to the success of the emerging cloud architecture. Bleikertz et al. [6] proposes one general differential approach to detect misconfigurations and security failures in virtualized infrastructure in near real-time. Their work primarily targets on verification of L2 policy. We provide a more comprehensive solution covering from L2 domain to L4 domain and we implemented it for real Openstack system. We traverse and model *iptables* for L4 state extraction, using approaches adapted from previous work [23, 4, 28].

The rest of this paper is organized as follows. Section 2 introduces the SDN functions in OpenStack and some inconsistency examples. Section 3 provides the overview of our methodology. Section 4, Section 5 and Section 6 introduces details of our methodology. In Section 7, we describe the system implementation. Then, Section 8 presents how our approach detects previous mentioned inconsistency cases effectively. Section 9 evaluates the performance of our system. Finally, Section 10 concludes the paper.

2. BACKGROUND

2.1 SDN in OpenStack

State inconsistency is a problem that can potentially arise in any edge-based SDNs. However, the design and implementation of the solution are dependent on the specific SDN environment. In this paper, we use OpenStack as the target environment to illustrate our approach. As a background, here we briefly introduce OpenStack and its SDN components.

A typical edge-based SDN setup in OpenStack involves three types of nodes (see Fig. 1): *Controller node*, which handles user requests for defining network services; *Compute node*, which is the end host that runs hypervisor to host VMs and implements the virtual network configurations for these VMs; *Network node*, which serves multiple roles, including virtual routers, DHCP servers, etc., in order to support the communications between different virtual networks. In a cloud data center, these different types of nodes are typically connected by hardware switches. The virtual network functions are defined by the user at the controller via API calls, and then communicated to the compute or network nodes, via AMQP messages, for actual configuration.

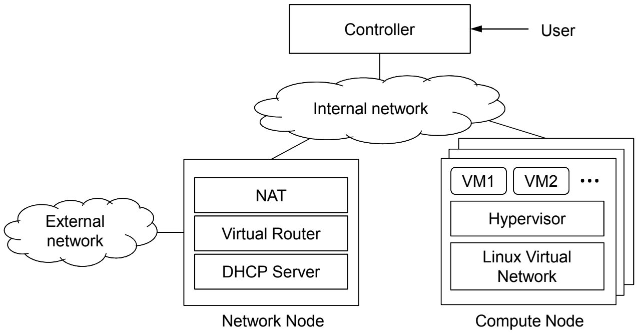


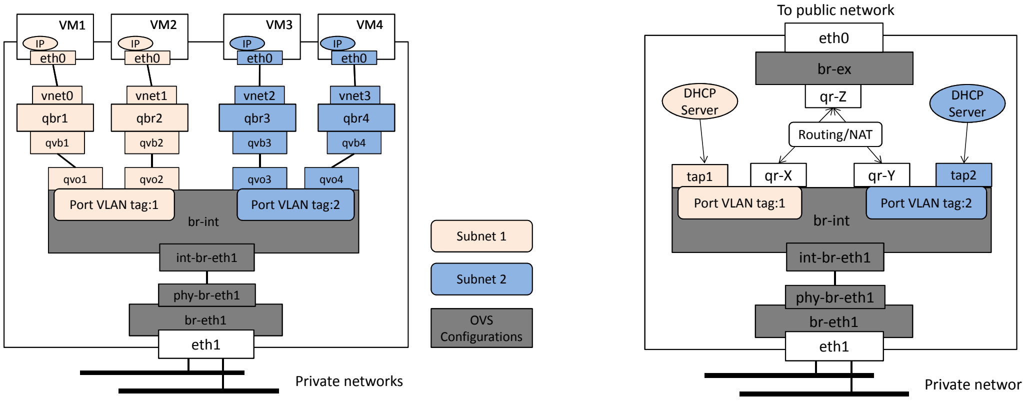
Figure 1: SDN components in OpenStack

The controller mainly allows user to define 1) Layer 2 networks; 2) Layer 3 subnets that are associated with a block of IP addresses and other network configurations, e.g., default gateways or DHCP servers; 3) virtual interfaces that can attach to a Layer 2 network as well as an IP address; 4) VMs that are attached to designated networks; 5) Layer 4 security group rules that determine the protocols and ports of the traffic admissible to selected VMs. These high-level network function definitions are stored in a central database.

The network functions defined on the controller are eventually translated into system configurations on the compute or network nodes, implemented using different technologies, e.g., Open vSwitch (OVS) as shown in Fig. 2.

L2 network function is implemented as internal VLANs[1] by OVS. On a compute node, VMs in the same L2 network are attached to the same VLAN via vNICs, Linux kernel devices, bridges, etc. For example, in Fig. 2, VM1 has a vNIC *eth0* connecting to a TAP device *vnet0*, which connects to a virtual ethernet interface pair *qvb1* and *qvo1* through Linux bridge *qbr1*, and then connects to VLAN 1. Across compute nodes, VMs attached to the same VLAN are connected to each other via a private network, which is either a switch-configured L2 network, or IP tunnels.

[1] Note these are internal VLANs defined in Linux OS, which are different from the VLANs configured on physical switches.



(a) Compute Node

(b) Network Node

Figure 2: Sample SDN configurations in OpenStack compute and network nodes.

L3 network functions are implemented mostly on the network node. The network node provides DHCP services to each L2 network, and assigns IP address to VMs. Through OVS, it also defines the routing between different subnets, as well as between subnets and the public network. The routing function also supports network address translation (NAT) for different subnets to communicate with each other.

L4 security groups are implemented as *iptables* rules in compute nodes, which include subset of rules for that accept or reject certain types of traffic from/to each VM.

2.2 Inconsistency Examples

In Openstack, we do observe occurrence of state inconsistency. When inconsistency occurs, the impact can be quite significant. In the following, we introduce three inconsistency examples to illustrate how state inconsistency could happen and how it affects the users.

2.2.1 L2 State Inconsistency Caused by Communication Error

In OpenStack, the controller communicates the network configurations to compute or network nodes via asynchronous AMQP messages. Occasionally, we observe that these messages can get lost, due to message queue overflow, TTL expiration, or network disruptions. When this occurs, state inconsistencies may appear, since the controller state in database has been updated, while the end hosts did not modify their configurations. For the example in Fig. 2(a), if the user requests through the controller to move VM1 from network 1 to network 2, but the message from the controller to the compute node is lost, then L2 inconsistency related to VM1 happens.

2.2.2 L3 State Inconsistency Caused by Software Bug

We also found there are software bugs in the current OpenStack code that can potentially lead to network state inconsistencies. An example we observed was depicted in Fig. 3 when we configure the virtual network using the *VlanManager* setting in OpenStack's *nova-network* functions.

Unlike in OpenStack *quantum* networks, in *nova-network*, there is no network node acting as virtual router. Instead, each compute node will implement the routing functions for the VMs running on it. In Fig. 3, VM1 and VM2 are on one compute node, VM3 and VM4 are on the other compute node. VM1 and VM3 are attached to *vlan 1*, while VM2 and VM4 are attached to *vlan 2*, through the Linux virtual devices and bridges (e.g., VM1 connects to *vlan1* through *vnet0* and *br100*). Across compute nodes, VMs on the same VLAN are connected through the physical switch, which assigns ID 1 and 2 to the two VLANs, respectively. For VMs on the same VLAN but different compute nodes to communicate, they go through their respective bridge gateways, e.g., VM1 will use *br100* (10.0.0.1) to reach *br100* (10.0.0.2), then reach VM3. For VMs on different VLANs to communicate with each other, they go through the local routes first. For example, for VM1 to reach VM4, it goes through VM2 on the same compute node to reach VLAN 2, through the local routing configuration, which allows packets from network *10.0.0.x* to be routed to network *10.1.0.x*.

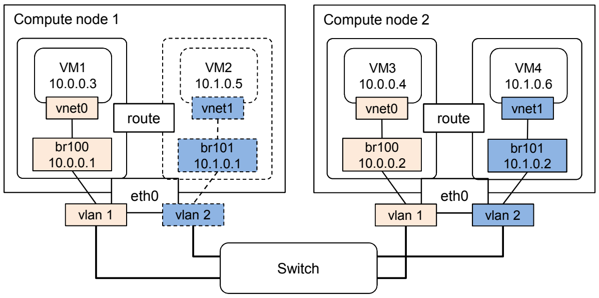


Figure 3: An example of L3 state inconsistency when using OpenStack *nova-network* function

However, in OpenStack *nova-network* implementation, the routing entries to a subnet are only created on the compute node, when a VM on that subnet is instantiated on this compute node. In the case of Fig. 3, if VM2 is not yet created on compute node 1, VM1 will not be able to reach any VM on VLAN 2, e.g., VM4, even though from the controller configuration, one would think there is a route set from VLAN 1 to VLAN 2. Thus, L3 inconsistency would occur under such condition.

2.2.3 L4 State Inconsistency Caused by Human Error

In this case, an operator mistakenly deleted the following ingress *iptables* rules for a VM on a compute node, when performing regular maintenance tasks:

```
...
DROP udp 0.0.0.0/0 0.0.0.0/0 udp spt:67 dpt:68;
RETURN 0.0.0.0/0 0.0.0.0/0;
...
```

The `DROP` rule ensures that any UDP packets from port 67 to port 68 will be rejected for this VM. Then the `RETURN` rule will allow the packets to traverse the calling *iptables* chains. In the effort of manually recovering the configurations for this VM, the operator accidentally switched the order of these two rules. As a result, the `DROP` rule becomes ineffective, as the packets will hit the `RETURN` rule first. Consequently, this VM will be exposed for potential security risks from this UDP port.

3. SYSTEM OVERVIEW

To address above inconsistency issues, we propose an approach to systematically identifying the state inconsistencies between controller and end hosts. As depicted in Fig. 4, our approach involves following steps:

- *Data Extraction*: Verification process starts with extracting network configuration data from both controller and end hosts. In controller, configuration data are typically stored in a central database and we just need to fetch related information from that database. In end hosts, we deploy a light-weight agent on each end host to execute certain system commands or check content of some configuration files on each end host to extract data.

- *Network State Abstraction*: To characterize the network states for both controller and end hosts, we model states at three layers: *Layer 2 state*, which indicates the MAC-layer network each Virtual Machine (VM) connects to; *Layer 3 state*, which represents the IP level reachability between VMs, and *Layer 4 state*, which describes the sets of rules for accepting/rejecting packets of various protocols and ports for each VM.

- *Data Parsing*: After extracting data from both controller and end hosts, we do data parsing to obtain network states in above state abstraction format. In our design, all extracted data are sent to one verification server to do data parsing and the following verification. When cloud scale becomes large, huge raw data parsing may becomes the bottleneck. As shown later, for the most time-consuming L4 layer parsing, we use smart *L4 state cache* to speed up that process.

- *State Verification*: After data parsing, we get controller network state and the actual end-host network state. We can do *State Verification* to check the inconsistency among the two state expressions. For the most time-consuming L4 layer verification process, we develop *two-level verification* method to avoid unnecessary computations.

Among the above four steps, *Data Extraction* is kind of straightforward, we put details of that step in Appendix A. For the residual three steps, we describe them in more details in the following three sections.

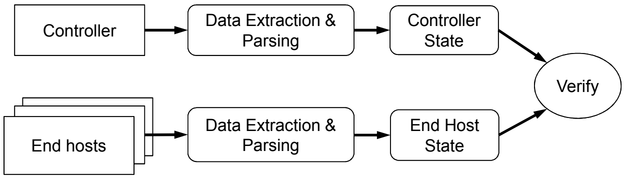


Figure 4: Overall approach for state inconsistency identification

4. NETWORK STATE ABSTRACTION

We characterize the network states for both controller and end hosts using a common format. Since SDN allows configuration of network functions at three different layers, we define network states correspondingly.

Layer 2 State: defines VM's MAC layer connectivities, i.e., whether a VM can receive ethernet packets from certain L2 network. We define the L2 state as mapping between the two:

$$\text{Map}_{l2} = \{\text{MAC}_i : \text{Network}_j\} \qquad (1)$$

where MAC_i represents the MAC address of a VM's vNIC, and Network_j represents the L2 network (uniquely identified by an ID, e.g., external VLAN ID) it is attached to.

Layer 3 State: defines the IP layer reachability between VMs, and between a VM and the external network. We define the connectivity within the private network as a binary matrix

$$\begin{array}{c|cccc} & \text{IP-MAC}_1 & \text{IP-MAC}_2 & \dots & \text{IP-MAC}_M \\ \hline \text{IP-MAC}_1 & r_{11} & r_{12} & \dots & r_{1M} \\ \text{IP-MAC}_2 & r_{21} & r_{22} & \dots & r_{1M} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{IP-MAC}_M & r_{M1} & r_{M2} & \dots & r_{MM} \end{array} \qquad (2)$$

If a VM with IP and MAC address combination, IP-MAC_i can reach another VM with IP-MAC_j, then $r_{ij} = 1$; otherwise, $r_{ij} = 0$. Note that because the same IP address can be reused in different private networks, we need to use both IP and MAC addresses to uniquely identify a VM's vNIC at layer 3. A VM can connect to the external network if and only if it is assigned with a public IP by the NAT router. Therefore, we can represent VMs' connectivity to the external network as the following mapping:

$$\text{Map}_{public\ l3} = \{\text{IP-MAC}_i : \text{Public IP}_j\} \qquad (3)$$

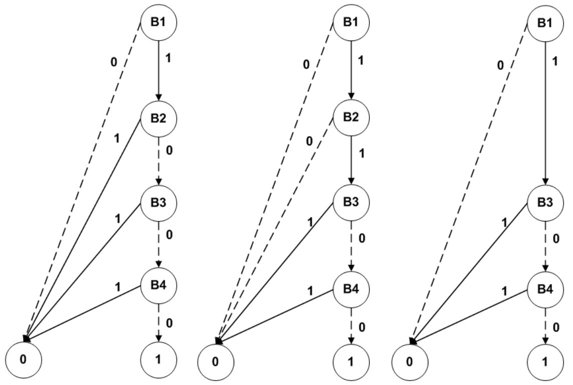
where Public IP_j represents the public IP address that a VM is NATed to, if it can reach the external network.

Layer 4 State: defines each VM's security groups and the associated packet filtering rules. We use Binary Decision Diagram (BDD) [7] as a representation of L4 state. Specifically, for each *iptables* rule, we generate a bitmap representation, which consists of five fields, with a total of 98 bits: *source IP range* (32 bits), *destination IP range* (32 bits), *protocol* (2 bits), *source port* (16 bits), and *destination port* (16 bits). This bitmap can then be presented in the form of a BDD [7], which compactly and uniquely represents a set of boolean values of these fields, and the corresponding actions when the values of these fields match certain conditions. As the examples shown in Fig. 5, a BDD has two end nodes: 0 represents the reject action, 1 represents the accepted action. B_n represents the nth bit that needs to be checked. Using this structure, IP address set $128.0.0.0/28$ and $192.0.0.0/28$ can be simply represented as Fig. 5(a) and Fig. 5(b), respectively.

Note that with BDD, we can also easily perform set operations, such as *Union*, *Intersection*, and *Difference*, which are important for this study. For example, to obtain the union of the two BDDs in Fig. 5(a) and 5(b), one can simply remove B_2, as the rest of the two BDDs are the same, as shown in Fig. 5(c). The union and intersection operations are needed when we analyze the aggregate effect of of multiple *iptables* rules, each represented in a BDD. The difference operation is important when comparing the BDD representations of L4 states between controller and end hosts.

5. STATE PARSING

After data is extracted, the next step is to parse it into network state representations. State parsing for the controller is straightforward, since the network configurations extracted from controller database can be directly translated into network states. State parsing for the end hosts, however, requires more effort. We describe how we do state parsing for end hosts in the following.



(a) 128.0.0.0/28 (b) 192.0.0.0/28 (c) 128.0.0.0/28 or 192.0.0.0/28

Figure 5: Examples of using Binary Decision Diagram representing layer-4 state

5.1 L2/L3 State Parsing

On a compute node, L2 state can be parsed by traversing the virtual device connections between each VM's MAC address and the internal VLANs configured on the hypervisor host. Across compute nodes, the internal VLAN will be mapped to an external L2 ID by Open vSwitch. Since the external L2 ID is assigned across compute nodes, we can use the VMs' associations to external L2 IDs to determine which VMs belong to the same L2 network. Note that here the external L2 ID will map to the network in Eq. (1).

L3 network states include both the IP-level reachability between VMs, i.e., Eq. (2), and their connectivity to the external networks, i.e., Eq. (3). For the former, we first check for each pair of VMs, whether they are connected in the same L2 network. If they are in the same L2 network, and their IP addresses are also configured to the same subnet, then they can reach each other. If they are not in the same subnet, we determine whether the two VMs use the same virtual router in the network node as gateway. If they do, then we check whether the virtual router is configured with routing entries that support the IP routing between the two IP subnets. For the latter, we need to check for each VM's private IP address, whether its gateway virtual router has the corresponding routing entry to allow it to reach a public IP address. In addition, we also need to check whether the virtual router implements NAT rules to translate between the two IP addresses. For this, we need to traverse the corresponding rules in *iptables*, following a procedure that will be discussed next.

5.2 L4 State Parsing

L4 state parsing involves analyzing each VM's security group configurations to generate the BDDs corresponding to its ingress and egress packet handling actions, respectively.

In end host, *iptables* rules are organized in chains. Each chain contains a sequence of rules, each of which defines a matching packet set, and the associated action *accept*, *drop*, *return*, or *call* another chain. Fig. 6 shows one example of *iptables* chain. Chain X is the main chain with default action *drop*. It calls chain Y at rule X_3, and another chain Z at rule X_4. Chain Y further calls chain J, and so on. We can characterize the calling relation between chains using a graph, in which each chain is represented by a node, and a directed link goes from X to Y, if chain X calls chain Y. Since there is no calling loop at the chain level, the calling relation graph is an acyclic graph. For example, Fig. 6 can be abstracted as a tree rooted at the main chain X.

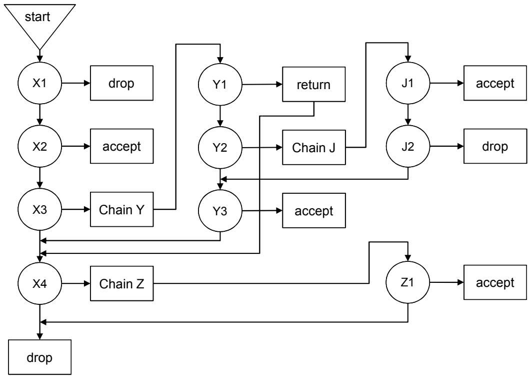


Figure 6: An example of *iptables* chains

To generate the BDD representation of a VM's L4 state, we need to traverse the entire *iptables* chain. We developed Algorithm 1 to parse all the rules of a chain sequentially to obtain the accepted/dropped packet sets (A/D), and the set of packets (R) triggering the action of returning to the calling chain, respectively. C denotes the set of packets that have not been matched by any rule yet, and is initialized to the set of all packets P. After parsing a new rule, which consists of the set of matched packets (M), rule action (*action*), and the chain to be called (*CalledChain*, if the action is '*call*'), the algorithm updates the unmatched packet set C, and adds the newly matched packets ($M \cap C$) to the set corresponding to the action type (line 5 to 11). If the action is to call another chain, the algorithm recursively calls itself to traverse the called chain and obtain its accept, drop and return sets to update the packet sets of the current chain (line 13-14). Since the calling relation between the chains is an acyclic directed graph, this algorithm can correctly traverse all chains in a set of *iptables* configurations. Note that using this algorithm, we only need to traverse each chain once, unlike the existing approaches [28], which typically require traversing a chain multiple times. For the *iptables* traversing in L3 State Parsing, we just need to modify Algorithm 1 to further consider the SNAT and DNAT packet sets.

Algorithm 1 ParseChain(*chain*)

```
1:  $A = D = R = \emptyset; C = P;$
2:  while chain is not empty do
3:      $(M, action, CalledChain)$=ReadNextRule();
4:      $E = M \cap C; C = C - M;$
5:      switch (action)
6:      case 'accept':
7:          $A = A \cup E;$
8:      case 'drop':
9:          $D = D \cup E;$
10:     case 'return':
11:         $R = R \cup E;$
12:     case 'call':
13:         $(A_1, D_1, R_1)$ = ParseChain(CalledChain);
14:         $A = A \cup (E \cap A_1); D = D \cup (E \cap D_1);$
15:     end switch
16: end while
17: return $(A, D, R)$
```

In a compute node, packets to/from a VM have to traverse *prerouting chain*, *forward chain*, and *post-routing chain* sequentially. In OpenStack implementation, VM's packet filtering rules are all placed in the forward chain, which consists of common subchains shared by all VMs, as well as VM-specific subchains, as illustrated in Fig. 7. The subchains for different VMs are uniquely identifiable by VM's physical device name or IP address.

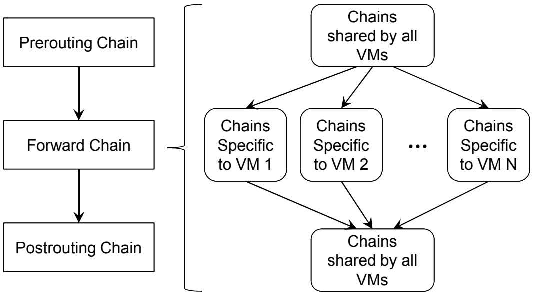


Figure 7: Structure of iptables chains on OpenStack compute node

Note that only VM-specific *iptables* rules are supposed to be modified by the controller during normal operations. However, on the compute node, there are other rules (e.g. those shared by all VMs in Fig. 7) that can affect individual VMs. We call these rules *shared* rules. The intersection between the two forms the overall L4 state of a VM. Unless we assume all *iptables* rules, including the shared rules, on compute node can only be modified by OpenStack controller and the modifications are always correct, we cannot ignore the possibility of shared rules being tempered. Therefore, we need to check the validity of shared rules as well. Specifically, we need to first parse out the BDD expression of the accepted packet set for the host by traversing all *iptables* rules. We then obtain the accepted packet set for a VM as a subset of the host's accepted packets that match the VM's device name.

5.3 L4 State Cache

Compared to L2, L3 state parsing, L4 state parsing is much more time-consuming. In L4 state, the network configurations are converted into BDD representations. We optimize this conversion process by implementing a cache.

Intuitively, if network configuration changes are small between two consecutive verifications, the later verification can reuse most of the BDDs generated in the previous rounds of verifications. Even if one runs the verification for the very first time, since most of L4 security group rules are common between the controller and end hosts, the BDDs generated from the controller can also be reused in end host state parsing. So the cache will keep a copy of BDDs, irrespective of whether they are generated from the controller or end hosts, as we parse the *iptables* rules. At the verification step, if BDD parsing is needed for a set of rules, the parser will first check whether the target rules already have BDDs in its cache. If yes, then the parser will avoid parsing the same rules again. To achieve the maximal gain, we design caches for both *individual rules* and *traversed chains*.

Caching for individual rules is straightforward. Whenever we encounter a rule of form $(S, action)$, where S is the string defining the matched packet set. We cache the BDD calculated for this rule, indexed by string S. We also cache (partially or fully) traversed chains. When traversing *iptables* chains, as described in Algorithm 1, we cache the intermediate results after parsing each additional chain. Each cache item is indexed by the ordered list of the rule strings (including all the string definitions S) in the corresponding traversal. For example, after traversing the *iptables* chains in Fig. 6, all individual rules in chain X, Y, Z, and J will be cached. BDDs for all partially and fully traversed chains are also cached, i.e., $Z = \{Z_1\}$, $\{J_1\}$, $J = \{J_1, J_2\}$, $\{Y_1, J\}$, $\{X_1, X_2, Y\}$ and $X = \{X_1, X_2, Y, Z\}$, etc.

With this caching mechanism, we can maximally reuse previous parsing results if only a small portion of the chains are modified. When the L4 state parser is requested to parse a new chain, it will first look up the full chain cache for exact match. Since a chain can call other chains, an exact match can be claimed for a chain only if all its descendant chains have exact match. In our implementation, we index all chains by topological ordering in the calling relation graph and always parse the chains with lower order first. So when we parse a chain, all its descendant chains must have been parsed. We can quickly check cache hit for this chain. If there is no exact match, it will then look for maximally matched partial chain, and only parse the BDD for the unmatched parts of the chain. If partial chain match cannot be found, it will look up the BDD cache for individual rules, then traverse the rest of the chain to calculate the BDD for the entire chain.

6. STATE VERIFICATION

Once the network states at L2, L3 and L4 are obtained, we can then verify the state consistency between controller and end hosts by comparing the two sets of states.

6.1 L2/L3 State Static Verification

L2/L3 State Verification: L2 states are represented as a set of mappings, as defined in (1), between VM's MAC address and L2 network. We can simply compare the two mapping sets, corresponding to the controller state and end host states, and identify different mappings.

Similarly, the L3 states of VM's reachability to public networks are also represented as a set of mappings, as described in Eq.(3). The state comparison also involves identifying different mappings between controller and end host states. The L3 states of inter-VM reachability are represented as a binary matrix, as stated in Eq.(2). We can compare the two matrices from controller and end hosts, to identify any entry r_{ij} that has different values.

6.2 L4 State Static Verification

L4 State Two-level Verification: L4 states are represented in BDDs that describe the ingress and egress packet filtering rules for each VM. If we obtain the BDDs for each VM from both the controller (BDD_E) and from the end hosts (BDD_H). The state comparison can be achieved by comparing their difference [7]:

$$\text{Diff}(\text{BDD}_E, \text{BDD}_H) = \text{BDD}_\triangle \tag{4}$$

If $\text{BDD}_\triangle = \phi$, then the states are consistent; otherwise, it represents the inconsistent L4 packet filtering behavior.

Note that it is not desirable to conduct BDD comparison in every round of L4 state verification, because deriving the BDDs for each VM involves traversing all of its *iptables* rules, which is computationally intensive. To reduce the computation overhead, we design a *two-level verification* approach like Algorithm 2 shows. The intuition behind the two-level verification approach is the following: if the OpenStack controller correctly configures the security group rules for VMs on a compute node, then the *iptables* rules should be configured following the specific template implemented by OpenStack. With this template, all shared *iptables* chains should follow a standard form, and the only variations are in the VM-specific chains, which define the security groups a VM is associated with. Plus, the VM-specific chains should contain the rules that are identical to the ones defined at the controller.

In practice, these assumptions should be true during most of the normal operations. And in these normal scenarios, we can first check whether the compute node's *iptables* configurations follow the OpenStack template, and then whether the rules for a VM match between controller and compute node. All these can be done at the string level, without involving BDD calculations. If both results are positive, then the L4 states are consistent between the two.

There are two anomaly cases that can potentially happen. The first case is the *iptables* configurations on compute node follow the OpenStack template, but the specific configurations of a VM are different from the ones at the controller. This could happen when there is communication error between the controller and compute nodes, or due to some (rare) software bugs, as we shall see later in our experiments. In this case, there will be VM rules mismatch between controller and compute node. Hence, we should invoke the BDD parsing for the VM-specific chains, and then compare these BDDs obtained from controller and compute node. Note that in this case, the BDD parsing only involves the chains specific to a VM with mismatched rules, not the other chains shared across VMs.

The second case is when the *iptables* configurations on compute node do not even follow the OpenStack template. This could happen when configurations are manually modified by system admin or by other program. In this case, we will have to generate the BDDs for all rules defined on the compute node, and then parse out the L4 state for each VM, using the method described in Section 5. Only by doing that can we fully verify the BDDs obtained from compute nodes against those from controller.

Algorithm 2 Two-level Verification

```
if iptables rules in end host follows OpenStack template then
    if rules for a VM match between controller and compute
    node in string level then
        L4 states are consistent for that VM
    else
        Invoke BDD parsing for VM-specific chains.
        Compare BDDs obtained from controller and end-host.
    end if
else
    Generate BDDs for all rules defined on compute node, parse
    out L4 state as described in Section 5.2.
    Compare BDDs obtained from controller and end-host.
end if
```

7. SYSTEM DESIGN AND IMPLEMENTATION

We designed and implemented a prototype of the proposed state verification system on OpenStack. Here we describe the implementation details of this prototype.

Following our overall methodology, the verification system primarily consists of three subsystems: a) *data collection* subsystem, b) *state parsing* subsystem and c) *state verification and reporting* subsystem, as shown in Fig. 8.

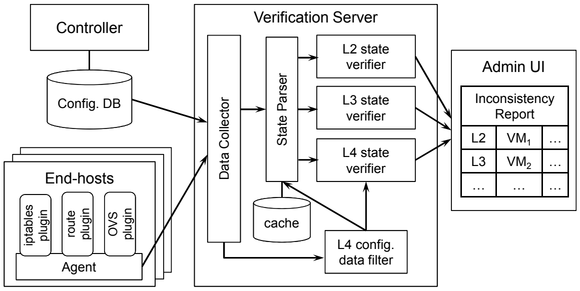


Figure 8: Prototype design and implementation

The data collection subsystem is responsible for collecting all the data required for state verification. It includes i) *data collection agent* residing on each end host, with plugins that can be invoked to collect various virtual network configurations, e.g., *iptables* rules, routing tables, etc., and ii) *data collector* residing on the verification server, which is responsible for receiving the data collected from the agents on the end hosts, as well as for issuing SQL queries to collect configuration data from the controller database.

The data collection agent is a lightweight program. We enable "File-Per-Table" mode [1] on the controller database, so that each table is stored in its own data file. The data collector will snapshot database table only when the corresponding data file is modified. The data collection agent on end hosts are invoked periodically as a `cron` job. To support continuous verification, we use NTP to ensure clock synchronization between these agents. Both the data collection agent and the central data collector are designed with a pluggable architecture. They can be easily extended to collect new data sources or data from other cloud management systems.

The data processing subsystem includes a state parser that ingests all the data from the data collector and parses it into the three layers of network state representations. Since L4 state parsing is a computation intensive task, we implement a configuration data filter to perform the *two-level comparison*, as described in Section 6. Only if the filter determines L4 state parsing is needed, will the state parser converts the configuration data into a BDD representation. To further reduce the overhead of L4 state parsing, the state parser stores previous parsing results in a local cache. Whenever a new L4 state parsing is requested, the cache is checked. If there is a hit in the cache, the state BDD will be directly retrieved from the cache. We describe the design of this cache later in this section. To support continuous verification, we need to do some additional processing, e.g., deleting duplicate state snapshots, etc.

Finally, the state verification and reporting subsystem takes the L2, L3, and L4 state representations obtained from both the controller and end hosts in each verification window, as well as generates inconsistency alerts. The state inconsistencies can be presented as an integrated part of an admin UI (e.g., the Horizon dashboard of OpenStack).

All components described above are implemented in Python. Except for the end host data collection agent and the state inconsistency reporting function, all other components run on a central verification server, which is hosted in the management network of OpenStack.

8. CASE STUDIES

In a production cloud, when the state inconsistency occurs, the impact can be quite significant. Previously in Section 2.2, we listed three inconsistency examples across L2, L3 and L4 layers. Here we show how our system can help identify those inconsistencies.

- *L2 State Inconsistency Caused by Communication Error in Section 2.2.1*: Our system will show that the L2 state at the controller is:
  {VM1 : Network 2}
  {VM2 : Network 1}
  {VM3 : Network 2}
  {VM4 : Network 2}

  However, the state parsed from the compute node would show that VM1 is still attached to network 1, instead of network 2. Thus, admin would be notified inconsistency happens.

- *L3 State Inconsistency Caused by Software Bug in Section*

2.2.2: On the controller, the L3 state between VMs indicates:

$$\begin{array}{cc} & \begin{array}{ccc} \text{VM1} & \text{VM3} & \text{VM4} \end{array} \\ \begin{array}{c} \text{VM1} \\ \text{VM3} \\ \text{VM4} \end{array} & \left(\begin{array}{ccc} - & r_{1,3} = 1 & r_{1,4} = 1 \\ r_{3,1} = 1 & - & r_{3,4} = 1 \\ r_{4,1} = 1 & r_{4,3} = 1 & - \end{array} \right) \end{array}$$

However, using our approach, the L3 state parsed from compute node 1 and 2 would show that $r_{1,4} = 0, r_{4,1} = 0$, hence there are inconsistencies between the two.

- *L4 State Inconsistency Caused by Human Error in Section 2.2.2*: Through our method, BDD_E derived from the controller and BDD_H from the compute node are clearly different: $\text{Diff}(\text{BDD}_E, \text{BDD}_H) = \text{BDD}_\triangle$, where $\text{BDD}_\triangle$ represents the ineffective `DROP` rule.

9. PERFORMANCE EVALUATIONS

To deploy our system in a production environment, we need to ensure it does not incur significant overhead to the underlying systems. We performed extensive experiments to measure our system's overhead. In this section, we report on some of the key results.

9.1 Experimental Setup

We set up a three-node OpenStack environment for our experiments: one node acting as both the controller and network node, the other two acting as compute nodes. Table 1 shows the hardware configuration of these three nodes.

Table 1: Hardware Configurations

Server	CPU	Memory
Controller/Network Node	Intel Core i3 3.07GHz 4MB Cache	4GB
Compute Nodes	Intel Core i3 3.30GHz 3MB Cache	8GB
Verification Node	Intel Core i3 2.50GHz 3MB Cache	8GB

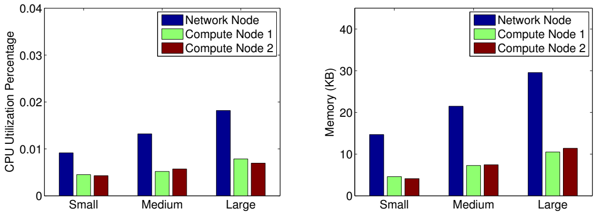
9.2 Overhead of Data Extraction

Table 2: Network Settings

Scale	# VM per node	# Security groups	# Rules per SG	# Subnet	# Floating IP	# Name space
Small	5	2	5	4	10	6
Medium	10	4	20	8	20	12
Large	15	6	50	12	30	18

As described in Section 7, data extraction from the end hosts involves invoking the agent on each network or compute node and executing system commands, which could incur some overhead. Here, we measure the extra CPU and memory consumption incurred from such data extraction over a one-hour period, with data extraction and parsing triggered every 10 minutes. To test the performance in virtual networks of different scales and complexities, we experiment with *small*, *medium* and *large* network configurations, with different number of VMs per compute node, security groups, etc., as shown in Table 2. Fig. 9(a) and 9(b) show the CPU and memory overhead. We observe that the data extraction agent imposes only a modest CPU overhead of $< 0.02\%$ and memory overhead of $< 30KB$ on end hosts, even in the most complex setting.

We also measure the data extraction time in different network settings in Table 3. For compute node, data extraction time increases with the number of VMs per node. The extracting time for the large setting with 15 VMs/node is about 0.5s. For network node, the extraction time is mostly dependent on the number of network name spaces, and the number of subnets and floating IPs, as they determine the number of networks and routing entries managed by the network node. As shown in the table, the extraction time is around 1.3s for the *large* network, with a fairly complex setting.



(a) Average CPU Overhead (b) Average Memory Overhead

Figure 9: Data extraction overhead caused by agents on end hosts

Table 3: Data Extraction Time

Scale	End-host Agent			Verification Server
	Network Node	Compute Node 1	Compute Node 2	Controller DB
Small	0.662s	0.228s	0.204s	0.020s
Medium	0.955s	0.346s	0.354s	0.022s
Large	1.314s	0.481s	0.522s	0.027s

Data extraction from the controller involves querying its management database. The overhead incurred by such query can be considered negligible for today's database servers. In our OpenStack environment, we use MySQL as the controller database. As shown in Table 3, the extraction time is less then 30ms in all the three settings.

Summary: The data extraction subsystem only incurs modest CPU and memory overhead to the SDN components, and can finish the extraction in seconds for typical virtual network settings.

9.3 Performance of Static State Verification

Next, we measure the time required for static state verification, after the configuration data has been collected in the verification server. As discussed in Section 5, the most time is spent on parsing the configuration data into network states, especially the L4 states. After parsing, the time needed for inconsistency checking is minimal. Table 4 shows the parsing and verification time at L2, L3, and L4 for the three network setups, where there was no state inconsistency.[2] Since we implemented the two-level comparison mechanism as described in Section 6, without any inconsistency, L4 parsing and verification finish after quick string-level checking. So the parsing and verification time are very small.

Table 4: Parsing and Verification Time

Scale	Parsing			Verification		
	L2	L3	L4	L2	L3	L4
Small	0.012s	0.006s	0.021s	<1ms	<1ms	<1ms
Medium	0.019s	0.014s	0.042s	<1ms	<1ms	<1ms
Large	0.029s	0.021s	0.093s	<1ms	<1ms	<1ms

When inconsistencies do exist and are detected by the string-level checking, we will have to verify L4 state using BDD. The caching mechanism in Section 5.3 can be used to speed up BDD calculation. To demonstrate the savings brought by two-level comparison and caching, we conduct four experiments with three different implementations:

1. *one-level BDD verification* that always generates BDD for all *iptables* chains and rules;

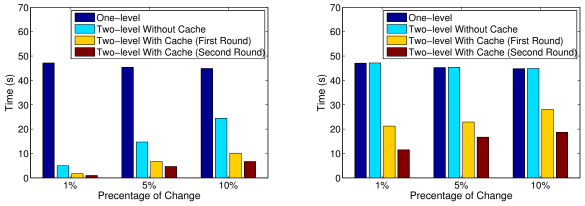
[2] **When inconsistency happens, parsing and verification time for L2, L3 have no change.**

2. *two-level verification without cache* that first checks if the end hosts' *iptables* rules match with those on the controller, and performs BDD parsing only if there is mismatch;

3. *two-level verification with cache* of the previously calculated BDDs, and run the program for the first time, and the cache is empty initially;

4. *two-level verification with cache*, same as in 3), but the program has been run once before, and the cache has been built from the previous verification runs.

We focus on the *large* network and run verification with three different levels of inconsistencies. These inconsistent configurations are injected by modifying the *iptables* rules on the compute nodes. By randomly selecting IP addresses and port numbers, we generate erroneous rules to replace randomly selected, original rules in *iptables*. We vary the modified configurations from 1%, 5%, to 10% of the overall configurations (measured by the number of *iptables* rules). In each setting, we also create two different scenarios according to the reasoning in Section 6: a) the modified rules only affect individual VMs, and b) the modified rules can affect both individual VMs and the entire compute node (i.e., all VMs on it).



(a) (b)

Figure 10: L4 state verification: a) when modifying rules for individual VMs; b) when modifying arbitrary rules

In the first scenario, we can quickly identify VMs whose rules are inconsistent at the string-level, and only need to do BDD parsing for those VMs. This explains the big saving of the two-level verification schemes over the one-level verification scheme in Fig. 10(a). Among the two-level verification schemes, caching results in additional savings. As expected, the second run can reuse BDDs calculated in the previous run, even with 10% rules changed between the two runs.

In the second scenario, the modifications are no longer associated with individual VMs. After inconsistencies are detected in the first-level verification, we will have to do comprehensive BDD parsing for each host. As a result, in Fig. 10(b), two-level verification consumes almost the same time as the one-level BDD verification. Nevertheless, caching can still reduce the time by reusing BDDs calculated earlier in the same run, or from the previous run.

Summary: Our system typically takes only seconds to perform state verification in a reasonably-sized SDN environment, hence can provide timely inconsistency alerts to the operator.

10. CONCLUSION

In this paper, we studied the problem of network state verification in OpenStack. Our solution consists of data extraction, state parsing and state verification at L2, L3 and L4 layers. To reduce the parsing time for L4 state, we proposed a two-level comparison design and developed a hierarchical BDD caching algorithm. Through experiments and emulation, we demonstrated that our verification system can effectively detect a variety of configuration inconsistencies with low computational overhead. The current approach detects inconsistencies by comparing two network state snapshots obtained from the controller and end hosts. In SDN, network configurations can change frequently. While the controller state can be captured easily, the actual state on end hosts may be "in transit" when the snapshot is taken. We are working on *continuous verification* that aligns snapshots taken on the controller and end hosts and then to identify legitimate transient snapshots on end hosts.

APPENDIX

A. DATA EXTRACTION

The verification process starts with extracting network configuration data from both controller and end hosts. Extraction from controller is straightforward: the configuration data is typically stored in a central database. For instance, *quantum* database table in OpenStack controller contains all network states that are supposed to be implemented. We can query the database to obtain the state information. For example, we can get each subnet's IP range and gateway's IP by running *SELECT network_id,cidr,gateway_ip FROM subnets*.

Extraction from the end hosts involves executing certain system commands or checking the content of some configuration files on each end host. In Table 5, we list the sources from which we extract end host configurations, when OpenStack *quantum* networking functions are used.

Table 5: Source of configuration data for OpenStack SDN functions

Source	Command / File Path	Network Layer
IPTable	iptables -t $table\_name$ -L -n	L4
Routing Table	netstat -rn	L3
Linux Bridge	brctl show	L2
Open vSwitch	ovs-dpctl show ovs-vsctl show ovs-ofctl dump-flows $bridge\_name$	L2
Veth Pair	ethtool -S $device\_name$	L2
Network Namespace	ip netns ip netns $ns\_name\ command$	L2, L3
VM Info	virsh list –all virsh dumpxml $domain\_name$	L2, L3, L4
IP Address	ip addr	L2, L3

For L2 state, we need to determine which network a VM's vNIC is attached to. Data extraction agent on each compute node will execute the related commands in Table 5, collecting the information regarding the vNIC, virtual devices, internal VLANs, and the connections among them. For L3 state, the agent extracts routing tables from the network node, Linux name space, VMs and their IP addresses from the compute nodes, etc. For L4 state, the raw data consists of *iptables* rules implemented in the OS of compute node. By executing the system commands as shown in Table 5, the agents can extract the rules applicable to each VM.

B. REFERENCES

[1] Innodb file-per-table mode.
`http://dev.mysql.com/doc/refman/5.5/en/innodb-multiple-tablespaces.html`.

[2] Messaging reliability/durability expectations.
`http://www.gossamer-threads.com/lists/openstack/dev/41915`.

[3] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, pages 37–44. ACM, 2010.

[4] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 17–31. IEEE, 1999.

[5] Bigswitch. Homepage. `http://www.bigswitch.com/`.

[6] S. Bleikertz, C. Vogel, and T. Groß. Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 26–35. ACM, 2014.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, pages 677–691, 1986.

[8] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A nice way to test openflow applications. *NSDI, Apr*, 2012.

[9] CloudStack. Homepage. `http://cloudstack.apache.org/`.

[10] Eucalyptus. Homepage. `http://www.eucalyptus.com/`.

[11] Forum. Floating IP Nat Error. `https://ask.openstack.org/en/question/48468/neutron-floating-ip-nat-on-wrong-router/`.

[12] Forum. VM doesn't get IP with Neutron. `https://ask.openstack.org/en/question/48147/vm-doesnt-get-ip-with-neutron-icehouse-centos-65/`.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011.

[14] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 55–60. ACM, 2012.

[15] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, et al. Leveraging sdn layering to systematically troubleshoot networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 37–42. ACM, 2013.

[16] X. Ju, L. Soares, K. G. Shin, and K. D. Ryu. Towards a fault-resilient cloud management stack. In *USENIX Workshop on Hot Topics in Cloud Computing(HotCloud)*, 2013.

[17] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[18] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. *NSDI, Apr*, 2012.

[19] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[21] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni. Testing amqp protocol on unstable and mobile networks. In *Internet and Distributed Computing Systems*, pages 250–260. Springer, 2014.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[23] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 177–187. IEEE, 2000.

[24] Openflow. Homepage. `https://www.opennetworking.org/`.

[25] OpenStack. Homepage. `http://www.openstack.org/`.

[26] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM, 2012.

[27] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A network-state management service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 563–574. ACM, 2014.

[28] L. Yuan and H. Chen. Fireman: a toolkit for firewall modeling and analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 199–213, 2006.

[29] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.