Identifying SDN State Inconsistency in OpenStack

Yang Xu, *Student Member, IEEE,* Yong Liu, *Member, IEEE,*
Rahul Singh, *Member, IEEE,* and Shu Tao, *Member, IEEE*

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 the high level network service definitions, which are maintained in the central controller, as state inconsistency often arises in practice due to unreliable state dissemination, 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.

Index Terms—OpenStack, Software Defined Networks, Network Configuration Verification.

1 Introduction

In Software Defined Networks (SDN), the control plane is decoupled from the data plane. Operators only need to 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 can 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 [4], [23]. In edge-based SDN, the 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 [24], CloudStack [8], Eucalyptus [9], etc.

A critical challenge in SDN is to ensure the consistency between high level network service definitions and low level configurations. In other words, how would an operator know the network services and policies defined at the controller are faithfully implemented by network devices? 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 errors during their operations. Misconfigurations on end hosts can potentially break the intended network functions. In a multi-tenant cloud, such misconfigurations 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 [10], [11]. In our own experience, those inconsistencies often arise due to the following reasons:

- *Unreliable State Dissemination*: In OpenStack, the controller sends network configurations to end hosts through asynchronous messages. If a message is lost (e.g., due to messaging server queue overflow) or communication is disrupted, the states of the two become out-of-sync [2]. Although one can use message acknowledgement to enhance reliability, end hosts may still fail to act on it properly after receiving the correct message. For example, the modification to *iptables*, *vSwitch*, etc, can be interfered by software configuration, access permission, or other softwares running on the same host, 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 human errors in virtual network configurations.
- *Software Bugs*: Even though rare, edge-based SDN implementations are not bug-free. We did experience cases in which the network configurations pushed into the end hosts do not exactly reflect the network policies defined at the controller. And some bugs are very hard to detect before the real deployment.

To address those problems, we propose an approach to systematically identifying the state inconsistencies between the controller and the end hosts. We model network states at three different network layers. In our approach, we will first extract the network configuration 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 then sent to a central verification server, which compares the two and identifies any potential inconsistencies between them.

Towards developing this approach, we made several contributions to address the following challenges:

- *Y. Xu and Y. Liu are with the Department of Electrical and Computer Engineering, New York University, Brooklyn, NY, 11201. E-mail: yx388@nyu.edu, yongliu@nyu.edu.*
- *R. Singh and S. Tao are with IBM T. J. Watson Research Center, Yorktown Heights, NY, 10598. E-mail: rahulsi@us.ibm.com, shutao@us.ibm.com*

Manuscript received April 19, 2005; revised September 17, 2014.

- *Network State Abstraction*: The first challenge is to develop a method to map the variety of virtual network configurations into a common network state abstraction. We developed succinct representations for network states at layer 2, layer 3, layer 4, and provided mechanisms to efficiently map raw configuration data into such state representations.
- *Sheer Volume of Data Processing*: A practical challenge is to deal with the sheer volume of configuration data to be processed and the network states to be interpreted. 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 (or more) of 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 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 similar configurations. Finally, we design a *two-level verification* method to further speed up the verification process.
- *Continuous State Verification*: If SDN converges to stable state at both controller and the end hosts, we could snapshot network states of these two to do static verification. However, network configurations tend to change frequently. The fact that the controller state and the actual state constantly evolve imposes another challenge: while the controller state can be captured easily, the actual state on end hosts may be "in transit" at the time when the snapshot is taken. We developed mechanisms to align snapshots taken on the controller and the end hosts and identify legitimate transient snapshots on the end hosts. *Continuous state verification* is done using these mechanisms.
- *System Design and Implementation*: The last challenge is to design and implement a verification system that can be used in a large scale production environment. To this end, 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 [20], [27]. Heller et al. [14] provided a survey on the existing studies and tools for troubleshooting SDN. Specifically, [12], [25] studied high level abstractions for easier OpenFlow configurations. The work in [3], [7], [18], [21] 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, Khurshid et al. [19] and Kazemian et al. [17] proposed to use trie structure or dependency graph to allow incremental checking. Also related are the work on debugging SDN configurations [13], [29].

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 doesn't have those 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. Bleikertz et al. [5] proposes a differential approach to detect misconfigurations and security failures in virtualized infrastructure in near real-time. Their work primarily targets on the verification of L2 policy. We provide a more comprehensive solution covering from L2 domain to L4 domain, and we implement it for real Openstack system. We traverse and model *iptables* for L4 state extraction, using approaches adapted from previous work [22], [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, Section 6 and Section 7 introduces details of our methodology. In Section 8, we describe the system implementation. Then, Section 9 presents how our approach detects previous mentioned inconsistency cases effectively. Section 10 evaluates the performance of our system. Finally, Section 11 concludes the paper.

2 Background

2.1 SDN in OpenStack

OpenStack is an open source Infrastructure-as-a-Service (IaaS) cloud management system that has gained industrywide support and adoption in recent years. It has several subsystems that are responsible for managing virtualized compute, storage, and network, respectively. Its networking subsystem, (known as Neutron), supports a variety of edge-based SDN options.

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 the actual configuration.

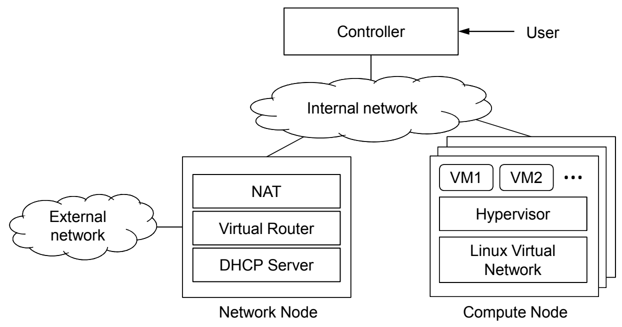
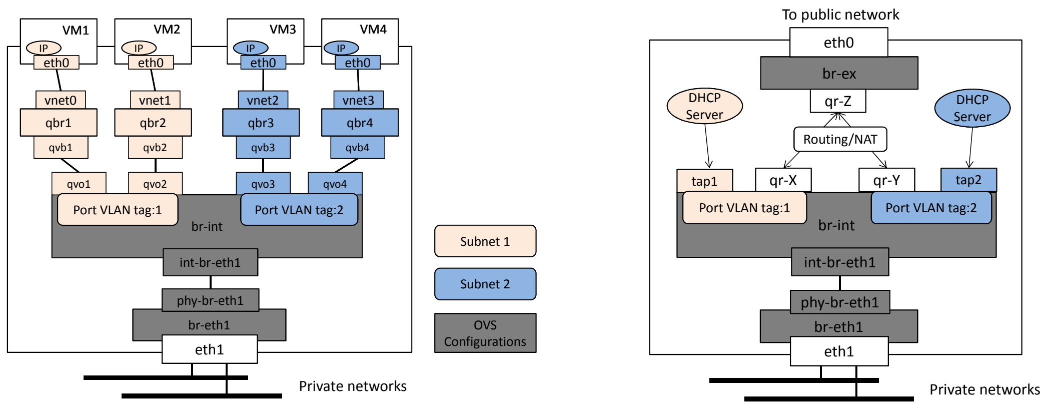


Fig. 1: SDN components in OpenStack



(a) Compute Node (b) Network Node

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

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. These configurations can be implemented with different technologies. For example, Open vSwitch (OVS) is one of widely used option in the community. Fig. 2 illustrates, together with OVS, how various Linux system configurations work together to implement the high-level network functions.

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.

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. Note that Linux supports multiple *name spaces* on a single OS, which allows the same IP address to be reused on multiple internal VLANs on the same compute node. Therefore, the routing function also supports network address translation (NAT) for different subnets to communicate with each other. NAT is also used to support the communication between private subnets and the public network, by translating between private IP and public IP (called floating IP) assigned to designated VMs.

L4 security groups are implemented as *iptables* rules in compute nodes, which include subsets of rules for accepting or rejecting certain types of traffic from/to each VM.

Obviously, in such an environment, the validity of network function critically relies on the configurations on the end hosts (both compute and network nodes) being consistent with the high level network state defined at the controller. Verifying such consistency is not a trivial task, given that so many configurable components on the end hosts are employed to realize the network functions.

2.2 Inconsistency Examples

In a real production environment, the state inconsistency does not occur often. But when it occurs, the impact can be quite significant. In Openstack, state inconsistencies can happen at all layers. In the following, we introduce an inconsistency examples across L2, L3 and L4 layers.

2.2.1 L2 State Inconsistency Caused by Unreliable State Dissemination

In OpenStack, the controller communicates the network configurations to compute or network nodes via asynchronous AMQP messages. Occasionally, we observe that end hosts fail to act on them properly, due to message loss, software configuration, permissions, etc. 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 is not executed correctly, 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 illustrates an example we experienced when configuring a virtual network using the *VlanManager* setting in OpenStack's *nova-network* [2] functions. In OpenStack *nova-network*, there is no network node acting as virtual router. Instead, each compute node implements the routing

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

2. Nova-network is the virtual networking option in older releases of OpenStack, which only allows VMs to be attached to switch-configured VLANs.

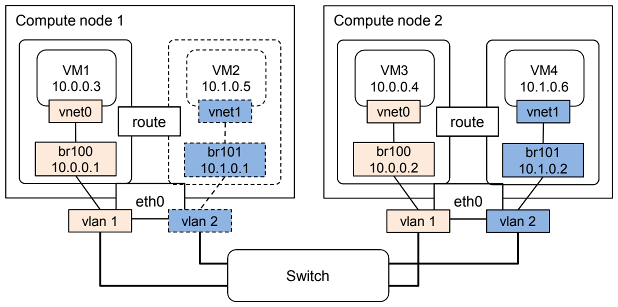


Fig. 3: An example of L3 state inconsistency when using OpenStack *nova-network* function

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 residing on different compute nodes to communicate, they go through their respective bridge gateways, e.g., VM1 will use *br100* (10.0.0.1) in Compute node 1 to reach *br100* (10.0.0.2) in Compute node 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*.

However, in OpenStack *nova-network* implementation, the routing entries to a subnet are created on the compute node only 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 assume there is a route from VLAN 1 to VLAN 2. Thus, L3 inconsistency occurs.

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

We propose an approach to systematically identify the state inconsistencies between the controller and end hosts. As depicted in Fig. 4, our approach involves the following steps:

- *Data Extraction*: In the controller, we fetch configuration data directly from a central database. In end hosts, we deploy a light-weight agent to execute system commands or read configuration files.
- *Network State Abstraction*: We model network states at three layers: *Layer 2 state*, which indicates the L2 network that each Virtual Machine (VM) connects to; *Layer 2 state*, which represents the IP level reachability between VMs and *Layer 4 state*, which describes the set of rules for accepting/rejecting packets of various protocols and ports on each VM.
- *Data Parsing*: We parse data extracted from the controller and end hosts to the above state abstraction format. All extracted data are sent to one verification server for parsing and verification. For the most time-consuming L4 layer parsing, we use efficient *L4 state cache* to speed it up.
- *State Verification*: After data parsing, we get the 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 a *two-level verification* method to avoid unnecessary computations.

In the following sections, we describe each of these steps in more details.

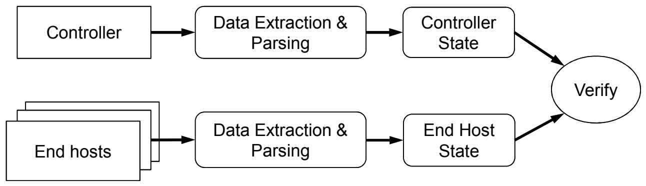


Fig. 4: Overall Approach

4 NETWORK STATE ABSTRACTION

A critical step of our approach is to characterize the network states for both the controller and the end hosts using a common format. We characterize the network states for both the controller and end hosts using a common format.

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

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

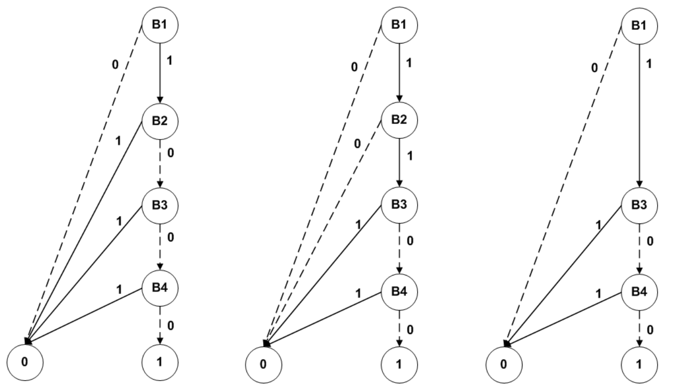
If a VM with IP and MAC address combination, IP-MAC$_i$ can reach another VM with IP-MAC$_j$, then $r_{ij} = 1$; otherwise, $r_{ij} = 0$. Since the same IP address can be reused in different private networks, we need to use $\langle IP - MAC \rangle$ tuple to uniquely identify a VM's vNIC at layer 3. 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:

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

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. 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 [6], 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 *reject*, 1 represents *accept*. B_n represents the nth bit that needs to be checked. Using BDD, IP address set 128.0.0.0/28 and 192.0.0.0/28 can be 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.



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

Fig. 5: Examples of BDD Representation of L4 State

Given the above network state abstractions, we next discuss the details on extracting, parsing the configuration data at each layer, and verifying state consistency between the controller and end hosts.

5 Data Extraction

The verification process starts with state data extraction from both the controller and the end hosts. Extraction from controller is straightforward: the configuration data is typically stored in a central database. For instance, the database named `neutron` in the OpenStack controller contains all the network states that are supposed to be implemented when *Neutron* network mode is used. In Table 1, we list the table that need to be queried. And we just run simple SQL command to get the 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".

TABLE 1: Central DB For Neutron Mode

Table	Related Layer
ovs_network_bindings	L2
ipallocations	L3
floatingips	L3
ports	L2, L3
subnets	L3
securitygroupportbindings	L4
securitygrouprules	L4

Extraction from the end hosts are more complicated. It typically involves executing certain system commands or checking the content of some configuration files on each end host. In Table 3 we list the methods we use to extract end host configurations in the Neutron network settings of OpenStack. (The table for Nova-network can be found in Table 2.)

TABLE 2: Extracting Method For Nova-Network Mode

Component	Command / File Path	Related Layer
IPTable	iptables -t *table_name* -L -n	L4
Routing Table	netstat -rn	L3
Linux Bridge	brctl show	L2
VM Info	virsh list –all virsh dumpxml *domain_name*	L2, L3, L4
IP Address	ip addr	L2, L3
Linux Vlan	/proc/net/vlan/config	L2

TABLE 3: Extracting Method For Neutron Mode

Component	Command / File Path	Related 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. For the data extracted from the controller, this information is readily available as the database defines the associations between MAC addresses and subnets. For the data extracted from the end hosts, local agent will execute the related commands in Table 3 on each compute node, collecting the information regarding the vNIC, virtual devices, internal VLANs, and the connectivities among them.

For L3 state, the controller database provides information regarding each VM's private IP address, the subnet it belongs to, the gateway of the subnet, as well as whether or not the private IP address will be mapped to a public IP address to communicate with the external network. At the end hosts, in addition to the L2 configurations, we also need to extract routing table, `iptables` in different Linux name space from the network node, VMs and their IP addresses from the compute nodes, etc.

For L4 state, the controller database contains the detailed definitions of security groups and the associated `iptables` rules. At the end hosts, the same rules should present for each VM. In OpenStack, these `iptables` rules are implemented at the hypervisor OS. By executing the system commands as shown in Table 3, we can extract the rules applicable to each VM.

6 State Parsing

After data is extracted, the next step is to parse the data into the corresponding 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 the configurations data from end hosts into network state representations in the following.

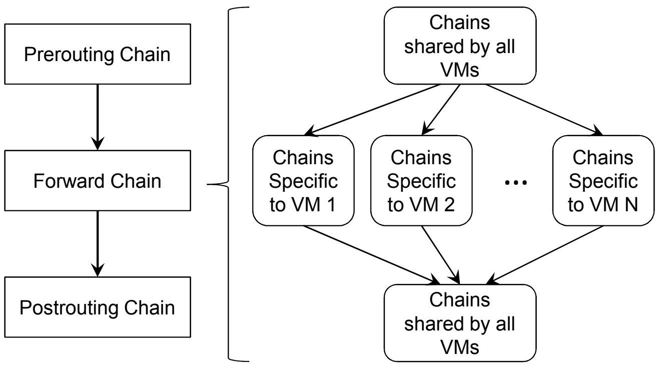


Fig. 6: Structure of iptables chains on OpenStack compute node

6.1 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 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.

6.2 L4 State Parsing

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

In end hosts, *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. 7 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. 7 can be abstracted as a tree rooted at the main chain X.

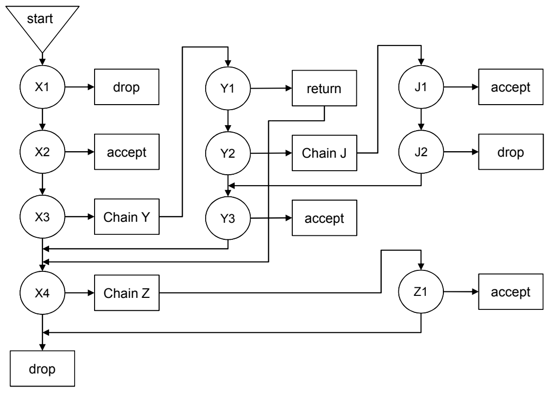


Fig. 7: An example of *iptables* chains

To generate 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.

In a compute node, packets to/from a VM have to traverse *pre-routing 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. 6. The subchains for different VMs are uniquely identifiable

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 $(A, D, R)$
```

by VM's physical device name or IP address. 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. 6) 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 the shared rules being tempered. Therefore, we need to check the validity of the 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.

6.3 L4 State Caching

In practice, converting L4 network configurations into raw BDD representation is time-consuming. We optimize the conversion process through caching. 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 rules to be parsed 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 a 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 traversed rule strings (including all the string definitions S). 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.

7 Static/Continuous Verification

7.1 Static 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.

7.1.1 L2/L3 Static 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 to identify any inconsistency. 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 inconsistency.

7.1.2 L4 Two-level Static Verification

L4 states are represented in BDDs that describe the ingress and egress packet filtering rules for each VM. After we obtain the BDDs for each VM from both the controller (BDD_E) and from the end hosts (BDD_H), their difference can be obtained as [6]:

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

If $\text{BDD}_\Delta = \phi$, then the states are consistent; otherwise, it represents the inconsistent L4 packet filtering behavior. Due to the computation overhead, it is not desirable to conduct BDD comparison in every round of L4 state verification. Instead, we design a *two-level verification* approach in Algorithm 2. The intuition 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 potential anomaly cases. 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 programs. 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 6.

Algorithm 2 Two-level Verification

```
1:  if iptables rules in end host follows OpenStack template then
2:      if rules for a VM match between controller and compute
        node in string level then
3:          L4 states are consistent for that VM
4:      else
5:          Invoke BDD parsing for VM-specific chains.
6:          Compare BDDs obtained from controller and end-host.
7:      end if
8:  else
9:      Generate BDDs for all rules defined on compute node, parse
        out L4 state as described in Section 6.2.
10:     Compare BDDs obtained from controller and end-host.
11: end if
```

7.2 Continuous Verification

When SDN states are static, we can use the above static state verification. However, configurations can change frequently in reality. Therefore, we need to continuously identify inconsistencies. For that we should continuously snapshot the configurations from the controller and end hosts. Thus, when to take a snapshot becomes an important design questions for our verification system. On the controller, since all legitimate state changes are triggered there, and all states are recorded in a central database, we can take a snapshot whenever the admin makes a change. On end hosts, the network configurations are collected by agents in a distributed manner. The complete state is pieced together using the snapshots taken from individual end hosts. Because end host states are collected from many configurations, taking snapshot for every detected change will result in unnecessarily high overhead. Therefore, we only take end host snapshots in a periodic fashion. As a result, the controller and end host snapshots are not taken at the same time. Furthermore, when a configuration change is triggered at the controller, it may take a variable amount of time to propagate to different end hosts. As a result, even with a legitimate network state change, the snapshot taken from end hosts may not match any snapshot from the controller, since the state of some end hosts could be "in transit" when the snapshot is taken.

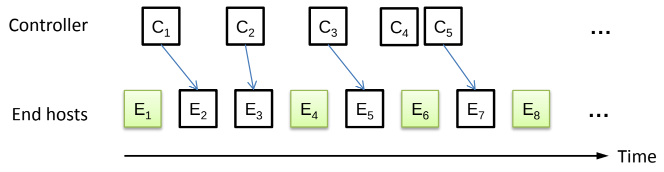


Fig. 8: Continuous state comparison between controller and end hosts

For instance, in Fig. 8, controller states C_1,C_2,C_3,C_5 match with end host states E_2,E_3,E_5,E_7, respectively. The other end host states are in-transit states. For example, E_4 is a temporary state between E_3 and $E5$, and does not match with any controller state. Also note that some controller state, e.g., C_4 may not have exact match with any end host state either, because the end host snapshot may not be taken right after the corresponding configurations are set.

In our design, we use a moving window to take a sequence of state snapshots from the controller and end hosts, respectively. With a same time window[3], we then compare the two to 1) determine the exact matches between controller state sequence $[C]$ and end host state sequence $[E]$, and 2) identify whether some of the unmatched end host states are in-transit state, or real inconsistency.

For 1), we use the following sequence alignment method adapted from [15], [26]. For controller state sequence $[C]$ and end host state sequence $[E]$, we seek the best matching subsequence $[C_1,C_2,...,C_i] \subset [C]$ and end host states $[E_1,E_2,...,E_j] \subset [E]$. We define the highest matching score between the two as $H_{i,j}$, which can be iteratively found using dynamic programming:

$$H_{i,j} = max \begin{cases} H_{i-1,j} - d \\ H_{i,j-1} - d \\ H_{i-1,j-1} + Match(C_i,E_j) \end{cases} \tag{5}$$

where $Match(C_i,E_j)$ denotes the matching score between snapshot C_i and E_j, d is a penalty for having one snapshot, either in $[C]$ or $[E]$, unmatched with any snapshot in the other sequence. We set $Match(C_i,E_j) = 0$ if C_i does not match E_j, $Match(C_i,E_j) = 2d$ if C_i matches with E_j and C_i was taken earlier than E_j, and $Match(C_i,E_j) = (-100)d$ if C_i matches with E_j, but C_i was taken later than E_j. The last case is when the state snapshot from the controller matches with an earlier state from end hosts, which is an obvious mismatching that should be avoided. Once we find $H_{i,j}$ for all possible subsequences of $[C]$ and $[E]$, we will pick the one with the highest score to be the best matching between $[C]$ and $[E]$. Note that this method assumes that the first snapshots in $[C]$ and $[E]$ are always a match, which can be guaranteed by setting our state comparison window to always start with the last matching snapshot in the previous window.

For 2), we use the following method. For L2 state, the controller can potentially change a VM's association from one L2 network to another. An in-transit end host state really means a VM's L2 association still reflects the previous controller state. For example, if L2 state sequences are represented in Fig. 8, if E_4 matches C_2, E_4 can be deemed an *in-transit* or *delayed* end

3. Here we assume the controller and compute nodes have loosely synchronized clocks.

host state. The same logic works for the VM's L3 association with public networks.

For L3 reachability between VMs, it boils down to evaluating the reachable IP address set for every private network, as we know the L2 association between VMs and the private networks already. In this case, to determine whether an end host state is in transit between two consecutive controller states, we evaluate whether this end host state (an IP address set) contains the intersect of configurations from two controller snapshots, but does not contain any configuration beyond their union. For instance, if Fig. 8 represents the L3 state for a private network, let us assume two controller states $C_2 = \{10.0.2.0/24, 10.0.3.0/24\}$ (reachable IP addresses) and $C_3 = \{10.0.3.0/24, 10.1.4.0/24, 10.0.5.0/24\}$. If end host snapshot $E_4 = \{10.0.2.0/24, 10.0.3.0/24, 10.1.4.0/24\}$, then it is an in-transit state between C_2 and C_3. However, if $E_4 = \{10.0.2.0/24, 10.1.4.0/24\}$, then it will be deemed an illegal state. The L4 in-transit state determination follows a very similar approach, except that the state is the accepted packet sets, represented as BDDs.

8 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.

Our 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. 9.

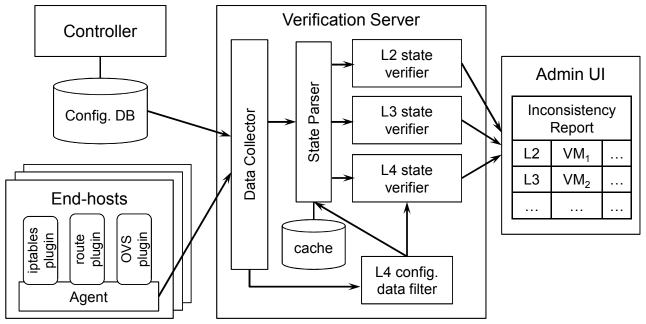


Fig. 9: Prototype design and implementation

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 used to receive the data collected from agents on end hosts, as well as to issue SQL queries to collect configuration data from controller database.

The data collection agent is a lightweight program. "File-Per-Table" mode [1] is enabled 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 to be able to extend to 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 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 7. 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 as described in Section 6.3. Whenever a new L4 state parsing is requested and there is a hit in the cache, the state BDD will be directly retrieved from the cache. 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 and the affected servers, VMs 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.

9 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 Unreliable State Dissemination 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:

$$\begin{array}{c} \\ \text{VM1} \\ \text{VM3} \\ \text{VM4} \end{array} \begin{array}{c} \begin{array}{ccc} \text{VM1} & \text{VM3} & \text{VM4} \end{array} \\ \begin{pmatrix} - & r_{1,3} = 1 & r_{1,4} = 1 \\ r_{3,1} = 1 & - & r_{3,4} = 1 \\ r_{4,1} = 1 & r_{4,3} = 1 & - \end{pmatrix} \end{array}$$

  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 2.2.3*: Through our method, BDD_E derived from the controller and BDD_H from the compute node are clearly different: $\text{Diff}(BDD_E, BDD_H) = BDD_\Delta$, where BDD_Δ represents the ineffective `DROP` rule.

10 Performance Evaluations

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. Machine configurations are reported in Table 4.

TABLE 4: 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

10.1 Overhead of Data Extraction

TABLE 5: Network Settings

Scale	# VM per node	# SG	# 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

TABLE 6: 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

We first measure the CPU and memory overhead of 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 numbers of VMs per compute node, security groups, etc., as shown in Table 5. Fig. 10(a) and 10(b) show the CPU and memory overhead. We observe that the data extraction agent imposes only light 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 6. Data extraction time from compute node and network node is acceptable even for the large network with a fairly complex setting. The data extraction time from the controller MySQL database is less then 30ms in all the three settings.

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

10.2 Static State Verification

Next, we measure the time required for static state verification, after configuration data has been collected in the verification server. Table 7 shows the parsing and verification time at L2, L3, and L4 for the three network setups, where there was no state inconsistency. [4] Since we implemented the two-level comparison mechanism in Section 7, without any inconsistency, L4 parsing and verification finish after quick string-level checking. The parsing and verification time are very small.

4. Even when inconsistency happens, parsing and verification time for L2, L3 have no change.

TABLE 7: 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. 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) *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 up from the previous run.

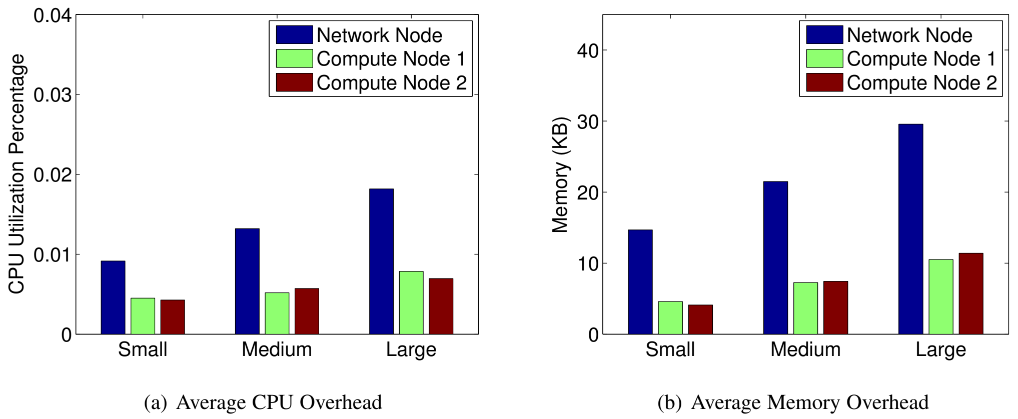
We focus on the *large* network and run verification with three different levels of inconsistencies. These inconsistent configurations are injected by modifying the *iptables* rules on the compute nodes. By randomly selecting IP addresses and port numbers, we generate erroneous rules to replace randomly 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 7: 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).

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. 11(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. In Fig. 11(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.

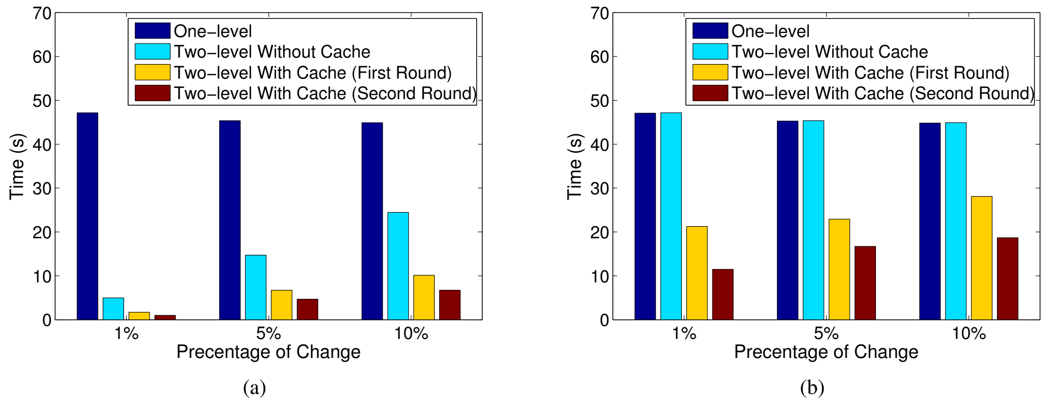
10.3 Continuous Verification

To validate the design of continuous verification mechanism, we set up experiments in a *medium* setting in Table 5. We randomly



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

Fig. 10: Data extraction overhead caused by agents on end hosts



(a) (b)

Fig. 11: L4 state verification: a) when modifying rules for individual VMs; b) when modifying arbitrary rules

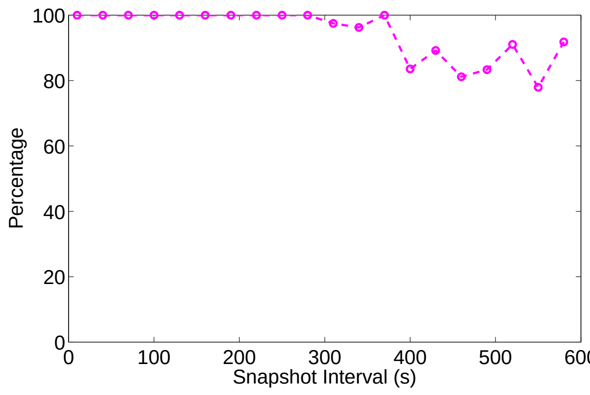


Fig. 12: Percentage of end host state snapshots correctly matched with controller states, with varying snapshot interval.

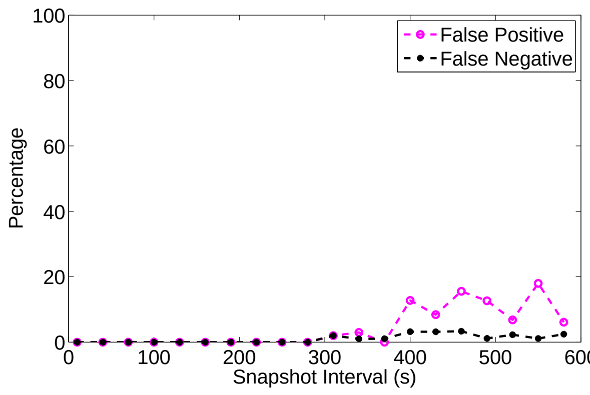


Fig. 13: Performance of identifying inconsistent states in continuous verification.

issue OpenStack API calls that can affect VMs' network configurations, e.g., adding/deleting VM, adding/deleting subnet, updating virtual router, updating security group, etc., to the system. We collect controller state upon every change and collect end hosts state once every 10 seconds. We run this experiment for 2 hours, during which 157 state changes were made from the controller.

We first evaluate system performance in an inconsistency-free environment, using the traces collected in the above experiment. We downsample the end host snapshots to emulate the setting that end host state snapshots are taken at longer intervals. Using the snapshot sequences from this 2-hour window, we first evaluate how well the mechanism introduced in Sec 7.2 finds the correct match between snapshots taken on controller and end host, while varying the snapshot interval. Fig. 12 shows the percentage of end host state snapshots that can be correctly matched with controller states. It is worth noting that unless the snapshot interval becomes too large (i.e., longer than 300s in our case), we can always match the end host states 100%. When snapshot interval is greater than 300s, performance start degradiing, as it becomes harder for the algorithm to find match for state snapshots from end hosts, given they are more sparsely taken.

To evaluate the effectiveness of our system in detecting state inconsistencies, we inject 21 inconsistent states, ranging from layer 2 to layer 4, to randomly selected end host state snapshots obtained in the previous study. We then invoke our verification process to identify these injected inconsistencies. As shown in Fig. 13, our method can identify 100% of these inconsistencies, as long as the end host snapshot interval is no larger than 300 seconds. When the snapshot interval is set larger, we will see some false positives and false negatives, largely due to the fact that the algorithm struggles to find the correct matching states between the controller and end hosts. Even in those situations, it can still identify more than 85% of state inconsistencies.

Summary: our continuous verification mechanisms can correctly detect most of the state inconsistencies in a given time window. To ensure good performance, we recommend setting end host snapshot interval no longer than twice the average time interval between consecutive controller state changes.

10.4 Performance Estimation

Currently we have no access to industry-scale Openstack deployment, but we can estimate the performance of our tools in large scale scenarios. In reality, if we use continuous verification, the modification between adjacent state snapshots is very minimal. When no inconsistency happens, parsing and verification process will be very quick, as Table 7 suggests. When inconsistency happens, the caching mechanism could help finish the most time consuming L4 verification within seconds if only few VMs have L4 inconsistency errors. Besides this, we could further utilize multiple verification nodes to speed up the whole process. Thus, it is estimated that our method satisfies the real world demand.

11 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. We further studied and proposed continuous verification schemes to handle the frequent SDN state changes. Through extensive experiments on our local testbed, we demonstrated that our verification system can effectively detect a variety of configuration inconsistencies in a dynamic cloud setting, with low computation and memory overheads.

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 *ACM workshop on Assurable and usable security configuration*, pages 37–44, 2010.

[4] Bigswitch. Homepage. http://www.bigswitch.com/.

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

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

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

[8] CloudStack. Homepage. http://cloudstack.apache.org/.

[9] Eucalyptus. Homepage. http://www.eucalyptus.com/.

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

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

[12] 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.

[13] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *ACM SIGCOMM HotSDN*, pages 55–60, 2012.

[14] 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 *ACM SIGCOMM HotSDN*, pages 37–42, 2013.

[15] D. G. Higgins and P. M. Sharp. Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988.

[16] X. Ju, L. Soares, K. G. Shin, and K. D. Ryu. Towards a fault-resilient cloud management stack. In *USENIX 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 NSDI*, 2013.

[18] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, pages 113–126, 2012.

[19] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX 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 *USENIX OSDI*, volume 10, pages 1–6, 2010.

[21] 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.

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

[23] Openflow. Homepage. https://www.opennetworking.org/.

[24] OpenStack. Homepage. http://www.openstack.org/.

[25] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, pages 323–334, 2012.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

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

[28] L. Yuan and H. Chen. Fireman: a toolkit for firewall modeling and analysis. In *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 *ACM CoNEXT*, pages 241–252, 2012.