TenantGuard: Scalable Runtime Verification of Cloud-Wide VM-Level Network Isolation

Yushun Wang*, Taous Madi*, Suryadipta Majumdar*, Yosr Jarraya†,
Amir Alimohammadifar*, Makan Pourzandi†, Lingyu Wang* and Mourad Debbabi*
*CIISE, Concordia University
Email: {yus_wang, t_madi, su_majum ami_alim, wang, debbabi}@encs.concordia.ca
†Ericsson Security Research, Ericsson Canada
Email: {yosr.jarraya, makan.pourzandi} @ericsson.com

***Abstract*—Multi-tenancy in the cloud usually leads to security concerns over network isolation around each cloud tenant's virtual resources. However, verifying network isolation in cloud virtual networks poses several unique challenges. The sheer size of virtual networks implies a prohibitive complexity, whereas the constant changes in virtual resources demand a short response time. To make things worse, such networks typically allow fine-grained (e.g., VM-level) and distributed (e.g., security groups) network access control. Those challenges can either invalidate existing approaches or cause an unacceptable delay which prevents runtime applications. In this paper, we present TenantGuard, a scalable system for verifying cloud-wide, VM-level network isolation at runtime. We take advantage of the hierarchical nature of virtual networks, efficient data structures, incremental verification, and parallel computation to reduce the performance overhead of security verification. We implement our approach based on OpenStack and evaluate its performance both in-house and on Amazon EC2, which confirms its scalability and efficiency (13 seconds for verifying 168 millions of VM pairs). We further integrate TenantGuard with Congress, an OpenStack policy service, to verify compliance with respect to isolation requirements based on tenant-specific high-level security policies.**

I. Introduction

The widespread adoption of cloud is still being hindered by security and privacy concerns, especially the lack of transparency, accountability, and auditability [1]. Particularly, in a multi-tenant cloud environment, virtualization allows optimal and cost-effective sharing of physical resources, such as computing and networking services, among multiple tenants. On the other hand, multi-tenancy is also a double-edged sword that often leads cloud tenants to raise questions like: "Are my virtual machines (VMs) properly isolated from other tenants, especially my competitors?" In fact, network isolation is among the foremost security concerns for most cloud tenants [2], [3], and cloud providers often have an obligation to provide clear evidences for sufficient network isolation [4], [5], either as part of the service level agreements, or to demonstrate compliance with security standards (e.g., ISO 27002/27017 [6], [7] and CCM 3.0.1 [8]).

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA
Copyright 2017 Internet Society, ISBN 1-1891562-46-0
http://dx.doi.org/10.14722/ndss.2017.23365

Verifying network isolation potentially requires checking that VMs are either reachable or isolated from each other exactly as specified in cloud tenants' security policies. In contrast to traditional networks, virtual networks pose unique challenges to the verification of network isolation.

- First, the sheer size of virtual networks inside a cloud implies a prohibitive complexity. For example, a decent-size cloud is said to have around 1,000 tenants and 100,000 users, with 17 percent of users having more than 1,000 VMs [9], [10]. Performing a cloud-wide verification of network isolation at the VM-level for such a cloud with potentially millions of active VM pairs using existing approaches results in a significant delay (e.g., Plotkin et al. [11] take 2 hours to verify 100k VMs). Most existing techniques in physical networks are not designed for such a scale, and will naturally suffer from scalability issues (a detailed review of related work is given in Section II) and quantitative comparison with state-of-the-art work is provided in Section VI.
- Second, the self-service nature of a cloud means virtual resources in a cloud (e.g., VMs and virtual routers or firewalls) can be added, deleted, or migrated at any time by cloud tenants themselves. Consequently, tenants may want to verify the network isolation repeatedly or periodically at runtime, instead of performing it only once and offline. Moreover, since any verification result will likely have a much shorter lifespan under such a constantly changing environment, tenants would naturally expect the results to be returned in seconds, instead of minutes or hours demanded by existing approaches [11].
- Third, a unique feature of virtual networks, quite unlike that in traditional networks, is the fine-grained and distributed nature of network access control mechanisms. For example, instead of only determined by a few physical routers and firewalls, the fate of a packet traversing virtual networks will also depend on the forwarding and filtering rules of all the virtual routers, distributed firewalls (e.g., security groups in OpenStack [12]), and network address translation (NAT), which are commonly deployed in a very fine-grained manner, such as on individual VMs. Unfortunately, most existing works fail to reach such a granularity since they are mostly designed for (physical)

network-level verification (i.e., between IP prefixes) instead of VM-level verification with distributed firewalls.

Motivating Example. Figure 1 shows the simplified view of a multi-tenant cloud environment.[1] The solid line boxes depict the physical machines (N compute nodes and one network node) inside which are the VMs, distributed firewalls (security groups), and virtual routers or switches. The virtual resources of different tenants (e.g., `VM_A1` of Alice, and `VM_B2` of Bob) are depicted by different filling patterns.

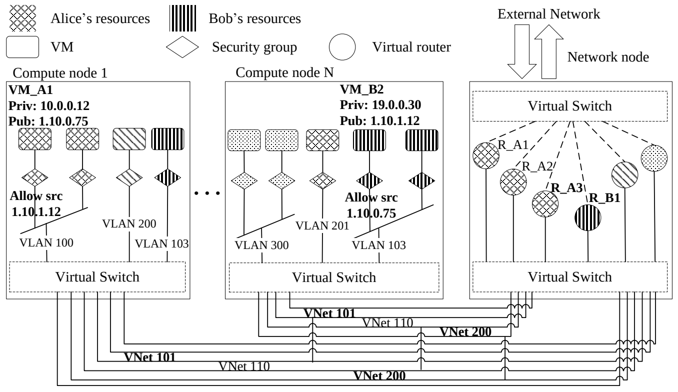


Fig. 1. An Example of a Multi-Tenant Cloud

- Network isolation may be compromised through either unintentional misconfigurations or malicious attacks exploiting implementation flaws. For example, assume the current security policies of tenants Alice and Bob allow their VMs `VM_A1` and `VM_B2` to be reachable from each other, as reflected by the two security group rules `allow src 1.10.1.12` and `allow src 1.10.0.75`. Now suppose Alice would like to stop accesses to her VM `VM_A1`, and therefore she deletes the rule `allow src 1.10.1.12` and updates her high level defined security policy accordingly. However, Alice is not aware of an OpenStack vulnerability OSSA 2015-021 [13], which causes such a security group change to silently fail to be applied to the already running VM `VM_A1`. At the same time, a malicious user of tenant Bob exploits another vulnerability OSSA 2014-008 [14] by which OpenStack (Neutron) fails to perform proper authorization checks, allowing the user to create a port on Alice's virtual router `R_A3` and subsequently bridges that port to his own router `R_B1`. Consequently, Alice's VM, `VM_A1`, will remain to be accessible by Bob, which is a breach of network isolation.

- To detect promptly such a breach of network isolation, the challenge Alice faces is again threefold. First, assume the cloud has $25,000$ active VMs among which Alice owns $2,000$. Since all those VMs may potentially be the source of a breach, and each VM may have both a private IP and a dynamically allocated public IP, Alice potentially has to verify the isolation between $25,000 \times 2,000 \times 2 = 100$ millions of VM pairs. Second, despite such a high complexity, Alice wants to schedule the verification to be performed every five minutes and is expecting to see the results within a few seconds, since she knows the result may only be valid until the next change is made to the virtual networks (e.g., adding a port by Bob). Finally, to perform the verification, Alice must collect information from heterogeneous data sources scattered at different locations (e.g., routing and NAT rules in virtual routers, host routes of subnets, and firewall rules implementing tenant security groups).

[1]To make our discussions more concrete, the examples will mostly be based on OpenStack, and Section VII discusses the applicability of our approach to other cloud platforms.

In this paper, we present *TenantGuard*, a scalable system for verifying cloud-wide, VM-level network isolation at runtime, while considering the unique features of virtual networks, such as distributed firewalls. To address the aforementioned challenges, our main ideas are as follows. First, TenantGuard takes advantage of the hierarchical structure found in most virtual networks (e.g., OpenStack includes several abstraction layers organized in a hierarchical manner, including VM ports, subnets, router interfaces, routers, router gateways, and external networks) to reduce the performance overhead of verification. Second, TenantGuard adopts a top-down approach by first performing the verification at the (private and public) IP prefix level, and then propagating the partial verification results down to the VM-level through efficient data structures with constant search time, such as radix binary tries [15] and X-fast binary tries [16]. Third, TenantGuard supports incremental verification by examining only parts of the virtual networks affected by a configuration change. Finally, TenantGuard leverages existing cloud policy services to check isolation results against tenant-specific high-level security policies. The following summarizes our main contributions:

- We propose an efficient cloud-wide VM-level verification approach of network isolation with a practical delay for runtime applications (13 seconds for verifying $25,246$ VMs and 168 millions of VM pairs, as detailed in Section VI).

- We devise a hierarchical model for virtual networks along with a packet forwarding and filtering function to capture various components of a virtual network (e.g., security groups, subnets, and virtual routers) and their relationships.

- We design algorithms that leverage efficient data structures, incremental verification, and an open source parallel computation platform to reduce the verification delay.

- We implement and integrate our approach into OpenStack [12], a widely deployed open source cloud management system. We evaluate the scalability and efficiency of our approach by conducting experiments both in-house and on Amazon EC2.

- We further integrate TenantGuard into Congress [17], an OpenStack policy checking service, in order to check the compliance of isolation results against tenants' predefined high-level security policies.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III describes the threat model and virtual network model. Section IV discusses our system design and implementation. Section V provides details on TenantGuard's integration into OpenStack and Section VI gives experimental results. Section VII discusses the adaptability and integrity preservation. Section VIII discusses limitations, provides future directions, and concludes the paper.

II. Related Work

Table I summarizes the comparison between existing works on network reachability verification and TenantGuard. The first column divides existing works into two categories based on the targeted environments, i.e., either cloud-based networks or non-cloud networks. The second and third columns list existing works and indicate their verification methods, respectively. The next column compares those works to TenantGuard according to various features, e.g., the support of parallel implementation, incremental verification, NAT, and all pairs reachability verification (which is the main target of TenantGuard). The next two columns respectively compare the scope of those works, i.e., whether the work is designed for physical or virtual networks, and whether it addresses control or data plane in such networks. Note that a L3 network is composed of a control plane for building the network typology and the routing tables based on various routing protocols (e.g., OSPF, BGP), and a data plane for handling packets according to the built routing tables. The last two columns show the size of input and verification time, respectively, as reported in those papers.

In summary, TenantGuard mainly differs from the state-of-the-art works as follows. First, TenantGuard performs verification at a different granularity level (i.e., all-pair VM-level vs single-pair router-level). Second, TenantGuard is more scalable (e.g., verifying 100k VMs within 17mins). Finally, TenantGuard employs custom algorithms instead of relying on existing verification tools (e.g., [11], [27], [29], [28]), which enables TenantGuard to more efficiently deal with complexity factors specific to the cloud network infrastructure such as a large number of VMs, longer routing paths (number of hops), and increased number of security rules.

Non-Cloud Network Verification. In non-cloud networks, several works (e.g., [30], [20], [18], [21], [19], [22], [23]) propose data plane analysis approaches, while others propose control plane analysis (e.g., [25], [24], [26]). Some existing works (e.g., [30], [20], [18]) address non-virtualized physical networks. Specifically, Xie et al. [30] propose an automated static reachability analysis of physical IP networks based on a graph model. Anteater [20] and Hassel [18] detect violations of network invariants such as absent forwarding loops. While those works are successful for verifying enterprise and campus networks, they cannot address challenges of large scale virtual networks deployed in the cloud with hundreds of thousands of nodes. For instance, Hassel [18] needs 151 seconds to compress forwarding tables before spending an additional 560 seconds in verifying loop-absence for a topology with 26 nodes.

Other works (e.g., [21], [19], [22]) propose approaches for virtualized networks. VeriFlow [21], NetPlumber [19] (extension of [18]), and AP verifier [22] outperform previous works by proposing a near real-time verification, where network events are monitored for configuration changes, and verification is performed only on the impacted part of the network. Those works propose query-based network invariants verification between a specific pair of source and destination nodes. In order to cover all-pairs, the total number of queries would grow significantly. This hinders the scalability of these approaches to tackle large cloud data centers. Furthermore, most of these works consider routers/switches as the source and destination nodes for their verification. NetPlumber and VeriFlow offer similar runtime performance. For a network of 52 nodes, Netplumber [19] checks all-pairs reachability in 60 seconds, whereas a single-machine implementation of TenantGuard takes only 4.6 seconds to verify a network of 4,300 nodes (see Figure 10). Libra [23] uses a divide and conquer technique to verify forwarding tables in large networks for subnet-level reachability failures. While Libra relies on the assumption that rules in switches consist of prefixes aggregating many subnets, we additionally deal with more specific rules (longer prefixes) by running the preorder traversal on the radix binary tries.

Works designed for control plane verification in physical networks like ARC [24], Batfish [25] and ERA [26], if applied to the cloud, would face the difficulty that (unlike physical networks) routing rules and ACLs for tenants' private virtual networks are not generated by the control plane.

Network Verification for Cloud Deployments. There are several works (e.g., [27], [11], [31], [32], [33], [28]) verifying the virtualized infrastructure in the cloud. Most of those solutions focus on verifying configuration correctness of virtualization infrastructures in terms of structural properties (e.g., Cloud Radar [28]), which is different from the properties targeted by TenantGuard. NoD [27], SecGuru [34] and their successor (Plotkin et al. [11]) are the closest works to TenantGuard, as they can check all-pairs reachability in physical networks for large cloud data centers. NoD is a logic-based verification engine that has been designed for checking reachability policies using Datalog definitions and queries. Plotkin et al. [11] improve the response time of NoD by exploiting the regularities existing in data centers lessen the verification overhead using bi-simulation and modal logic. The experimental results reported in Section VI show that TenantGuard outperforms those tools.

Congress [35], is an open project for OpenStack platforms. It enforces policies expressed by tenants and then monitors the state of the cloud to check its compliance. However, reachability requires recursive Datalog queries [35], which are difficult to solve and are not supported by Congress. Therefore, we integrated TenantGuard into Congress in order to check network isolation results provided by TenantGuard against tenants' security policies defined in Congress (Section V). Additionally, by integrating TenantGuard to Congress, we augmented Congress capabilities to support reachability-related policies as NoD without modifying Datalog-based policy language provided by Congress.

III. Models

In this section, we describe the threat model and propose a hierarchical model for cloud virtual networks.

A. Threat Model

Our threat model is based on two facts. First, our auditing solution focuses on verifying the security properties specified by cloud tenants, instead of detecting specific attacks

Network	Proposals	Methods	Features				Physical vs Virtual net.		Control vs Data plane		Size of input			Verif. Time
			Paral.	Incr.	NAT	All Pairs Reach.	Phys.	Virt.	Ctr.	Data	VMs	Routers	Rules	
Non-Cloud	Hassel [18]	Custom algorithms			●		●			●	-	26	756.5k	-
	NetPlumber [19]	Graph-theoretic	●	●	●	●		●		●	-	52	143k	60
	Anteater [20]	SAT solver	●		●		●			●	-	178	1,627	-
	Veriflow [21]	Graph-theoretic		●				●		●	-	172	5,000k	-
	AP verifier [22]	Custom algorithms		●	●			●		●	-	58	3,605	-
	Libra [23]	Graph-theoretic	●	●	●		●			●	-	11,260	2,650k	-
	ARC [24]	Graph-theoretic					●		●		-	few tens	-	-
	Batfish [25]	SMT Solver				●	●		●		-	21	-	86,400
	ERA [26]	Custom Algorithms					●		●		-	over 1,600	-	-
Cloud	NoD [27]	SMT Solver			●	●	●			●	100k	-	820k	471,600
	Plotkin et al. [11]	SMT Solver			●	●	●			●	100k	-	820k	7,200
	Cloud Radar [28]	Graph-theoretic						●			30k	-	-	-
	Probst et al. [29]	Graph-theoretic			●			●	●		23	-	-	-
	TenantGuard	Custom algorithms	●	●	●	●		●		●	100k	1,200	820k	1,055.88

TABLE I. COMPARING FEATURES AND PERFORMANCE OF DIFFERENT EXISTING SOLUTIONS WITH TENANTGUARD. THE SYMBOL (●) INDICATES THAT THE PROPOSAL OFFERS THE CORRESPONDING FEATURE. ALL VERIFICATION TIME MEASUREMENTS ARE REPORTED IN SECONDS.

or vulnerabilities (which is the responsibility of IDSes or vulnerability scanners). Second, the correctness of our auditing results depends on correct input data extracted from logs and databases. Since an attack may or may not violate the security properties specified by the tenant, and logs or databases may potentially be tampered with by attackers, our auditing results can only signal an attack in some cases. Specifically, the in-scope threats of our solution are attacks that violate the specified security properties and at the same time lead to logged events. The out-of-scope threats include attacks that do not violate the specified security properties, attacks not captured in the logs or databases, and attacks through which the attackers may remove or tamper with their own logged events. We assume each cloud tenant has defined its own security policies on network isolation in terms of reachability between VMs. We focus on the virtual network layer (layer 3) in this paper, and our work is complementary to existing solutions at other layers (e.g., verification in physical networks or isolation w.r.t. to covert channels caused by co-residency; more details are given in Section II). Finally, we assume the verification results (e.g., which VMs may connect to a tenant) do not disclose sensitive information about other tenants and regard potential privacy issues as a future work.

B. Virtual Network Model

Here, we define a hierarchical model to capture various components of a virtual network and their logical relationships. The following example provides intuitions on the model we propose.

Example 1: Figure 2 illustrates an instance of our model that captures the virtual networks of tenants Alice and Bob, following our example shown in Figure 1. Each tenant can create several subnets (e.g., `SN_A1` and `SN_A2` of Alice). A subnet (e.g., `SN_A2`) is generally associated with a CIDR (e.g., `10.0.0.0/24`) and a set of forwarding rules (host routes) specifying the default gateway (e.g., router interface `IF_A11`). A newly created VM (e.g., `VM_A1`, not shown here) will be attached to a virtual port (e.g., `VP_A1`) on a subnet (e.g., `SN_A2`) and associated with a private IP (e.g., `10.0.0.12`). Ingress and egress security groups are associated with the virtual ports of VMs and act as virtual firewalls. Routers (e.g., `R_A1`) interconnect different subnets to route intra-tenant (e.g., between `SN_A2` and `SN_A3`) and inter-tenant traffic and connect them to external networks (e.g., `ExtNet_1`) via router gateways (e.g., `RG_A1`). Several interconnected external networks (not shown in the figure) may exist, where each (e.g., `ExtNet_1`) can have a routable public IP address block (e.g., `1.10.0.0/22`). For inter-tenant traffic, at least one router from each tenant must be involved and the traffic generally traverses external networks. For any communication going through external networks, a public IP address is allocated per VM (e.g., `VP_A1.Public_IP=1.10.0.75`) depending on which external network (e.g., `ExtNet_1`) connects to the subnet of the VM (e.g., `SN_A2`). The mapping between private and public IP addresses is maintained through NAT rules at routers.

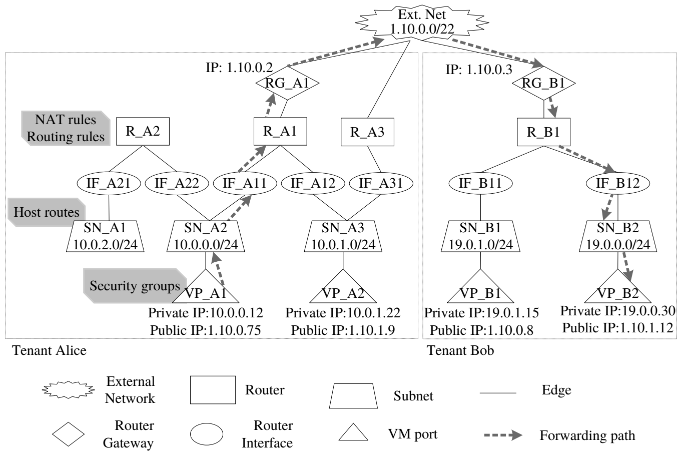


Fig. 2. An Example of the Virtual Network Model

More generally, Figure 2 may be abstracted as an undirected graph with typed nodes, as defined in the following.

Definition 1: A virtual network model is an undirected graph $G = (V, E)$, where V is a set of typed nodes each of which is associated with a set of attributes $s = \{id, tenant\_id, Public\_IP, Private\_IP, type, rules\}$, where $type \in \{vm\_port, subnet, v\_router, v\_router\_if, v\_router\_gw, ext\_net\}$, representing VM port, subnet, router, router interface, router gateway, and external network, respectively. E is a set of undirected edges representing the logical connectivity among those network components.

A virtual network model G can usually be decomposed into a set of maximally connected sub-graphs [36] (denoted by $C_i = (V_i, E_i)$ in later discussions) by removing all edges between router gateways and external networks. Those subgraphs represent different tenants' private virtual networks, which are connected to external networks via the removed edges. We will leverage this characteristic later in Section IV to tackle the complexity issues.

C. Forwarding and Filtering Model

In the following, we first model how packets may traverse a virtual network, and then formalize the network isolation property that we aim to verify.

Forwarding and Filtering. Network packets traversing virtual networks are typically governed by both filtering (security group rules) and forwarding (routing) rules, as demonstrated in the following example.

Example 2: Figure 2 shows a dotted line representing the sequence of edges traversed by a set of packets from `VM_A1` to `VM_B2`, which represents the forwarding path controlled by different nodes between both corresponding virtual ports. Packets sent to the virtual port (e.g., `VP_A1`) are processed by the egress security group rules then either dropped or forwarded to the subnet node `SN_A2`. According to host routes associated with `SN_A2` and the destination address (i.e., `VM_B2.Public_IP`), packets are either dropped or forwarded to the default gateway, which is the router interface `IF_A11` of the router `R_A1`. At the router node, packets' headers are matched with the routing rules, and are either forwarded to `RG_A1` and then to the associated external network `ExtNet_1`, or dropped. Packets destined to `VM_B2` are then forwarded by `ExtNet_1` to the router `RG_B1`. If matching forwarding rules for these packets are found at nodes `RG_B1`, `R_B1` and `SN_B2`, then the edges between `RG_B1` and `VP_B2` are traversed. At `VP_B2`, only packets matching the ingress security group rules are forwarded to their destination. Note that at the level of `R_A1` (resp. `R_B1`), packets are transformed using NAT rules by replacing the source (resp. destination) private (resp. public) IP of `VM_A1` (resp. `VM_B2`) with the corresponding public (resp. private) IP.

More generally, the following definition models the way packets traverse virtual networks using a forwarding and filtering function capturing respectively routing and security group rules.

Definition 2: **Forwarding and Filtering Function.** Given a virtual network model $G = (V, E)$,

- let $p \in \mathcal{P}$ be a symbolic packet (similarly as in [37]) consisting of a set of header fields (e.g., source and destination IPs) and their corresponding values in $\{0,1\}^L$ such that L is the length of the field's value, and
- let $(p,(u,v))$ be a *forwarding state* where (u,v) is the pair of nodes in G representing respectively the previous hop node (i.e., the sender node) and the current node (i.e., the node v where the packet is located in the current state).
- The forwarding and filtering function fd_G returns the successor forwarding states $\{(p'_i,(v,w_i))\}_{i\in\mathbb{N}}$, such that each $w_i \in V$ is a receiving node according to the results of rules matching at node v, and p'_i is the symbolic packet resulting from a set of transformations (e.g., NAT) over packet p before being forwarded to w_i where $\{v, w_i\}_{\forall i\in\mathbb{N}} \in E$.
- A forwarding path for packet p from node u to node v is a sequence of forwarding states $(p,(null,u))\cdots(p',(v,null))$.

As a convention, we will use $null$ in forwarding states to denote a forwarding state where the symbolic packet has been dropped (e.g., $(null,(w,null))$), a packet initially placed on a node v (e.g., $(p,(null,v))$), or a packet received by w after the last hop (e.g., $(p,(w,null))$).

Network Isolation. With the virtual network model and forwarding and filtering function just defined, we can formally model network isolation and related properties as follows.

Definition 3: Given a virtual network model $G = (V, E)$,

- for any u,$v \in V$, we say u and v are reachable if there exists a packet $p \in \mathcal{P}$ and a forwarding path for p from u to v. Otherwise, we say u and v are isolated.
- A forwarding loop exists between $u \in V$ and $v \in V$ if there exists $p \in \mathcal{P}$ destined to v and $w, w' \in V$ such that $(p,(w,w'))$ is a reachable forwarding state and that $fd_G((p,(w,w'))) = (p,(w',w))$.
- A blackhole exists between $u \in V$ and $v \in V$ if there exists $p \in \mathcal{P}$ destined to v and $w, w' \in V$ such that $(p,(w,w'))$ is a reachable forwarding state and $fd_G((p,(w,w'))) = (null,(w',null))$.

The properties given in Definition 3 can serve as the building blocks of any network isolation policies specified by a cloud tenant. The specific forms in which such security policies are given are not important, as long as such policies can unambiguously determine whether two nodes should be reachable or isolated. Therefore, our main goal in verifying a tenant's security policies regarding network isolation is to ensure any two nodes are reachable (resp. isolated) if and only if this is specified in such policies. In addition, our verification algorithms introduced in Section IV can also identify forwarding loops and blackholes as anomalies in virtual networks.

IV. TenantGuard Design and Implementation

In this section, we first provide an overview of our approach and then introduce the data structures and the verification algorthims in details.

A. Overview

Due to the sheer size of a cloud, verifying separately each pair of VMs (query-based approach) or directly computing all possible forwarding paths for all pairs of VMs (henceforth called the *baseline algorithm*) would result in an unacceptable response time, and not scale to large clouds, as will be demonstrated through experiments in Section VI. Also, the use of (possibly overlapping) private IPs and dynamically allocated public IPs in the cloud can make things even worse. To address those issues, TenantGuard leverages the hierarchical virtual network model presented in Section III-B by partitioning the verification task into a prefix-level verification followed by a VM-level verification. Prefix-level verification splits further the virtual networks into a set of private IP prefixes (i.e., tenants' subnets) and a set of public IP prefixes (i.e., external network IP prefixes), which results in a three-step approach, as it will be

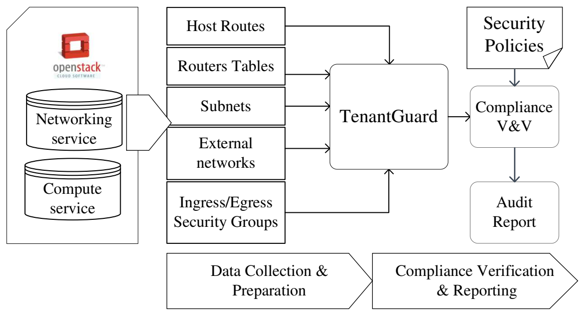


Fig. 3. An Overview of TenantGuard

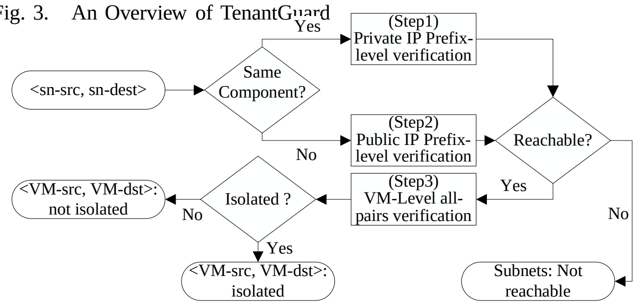

<sn-src, sn-dest>
Same Component?
Yes
(Step1) Private IP Prefix-level verification
No
(Step2) Public IP Prefix-level verification
Reachable?
Yes
No
(Step3) VM-Level all-pairs verification
Isolated ?
No
<VM-src, VM-dst>: not isolated
Yes
<VM-src, VM-dst>: isolated
Subnets: Not reachable

Fig. 4. A Flow Chart Illustrating Our Three-Step Approach

detailed later. Furthermore, we use efficient data structures that allow handling all-pair verification at once instead of a query-based approach. As we will confirm with experimental results in Section VI, those conceptual advances allow to scale to cloud-wide, VM-level verification of network isolation. Figure 3 provides an overview of the TenantGuard system. Input data from the cloud infrastructure management system, including router rules, host routes, and security groups, are collected and processed using efficient data structures as it would be detailed in Section IV-B. The preservation of collected data integrity is discussed in Section III-A. Once the verification results are returned, compliance verification compares such results with the tenant's pre-defined security policies. Finally, the corresponding auditing report is generated and presented to the tenant. Figure 4 provides a high-level flow graph corresponding to our three-step approach. Each element of the graph will be detailed in Section IV-C.

To grasp the intuition behind our three-step verification approach, we present an example.

Example 3: Figure 5 illustrates the application of our three-step verification approach using our running example shown in Figure 2. In `Step 1` (ref. Section IV-C1), prefix-level isolation verification within the same components/sub-graph using private IP is performed. For instance, the isolation between Alice's subnets `SN_A2` and `SN_A3` through the router `R_A1` is verified using their respective private IP prefixes (e.g., `10.0.0.0/24` and `10.0.1.0/24`). In `Step 2` (ref. Section IV-C1), prefix-level isolation verification between different components (e.g., `SN_A2` and `SN_B2`) is performed via each adjacent external network (e.g., `ExtNet_1`). This step is further decomposed into `Step 2.a` for verifying isolation between the source subnet (e.g., `SN_A2`) and the external network, and `Step 2.b` for verifying isolation between the external network and the destination subnet (e.g., `SN_B2`). This verification also involves public and private IP NAT. Finally, `Step 3` (ref. Section IV-C2) performs VM-level security groups verification for any pair of subnets found to be reachable using `Step 1` and `Step 2`.

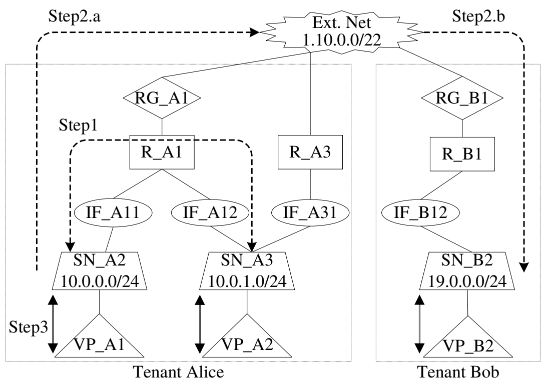
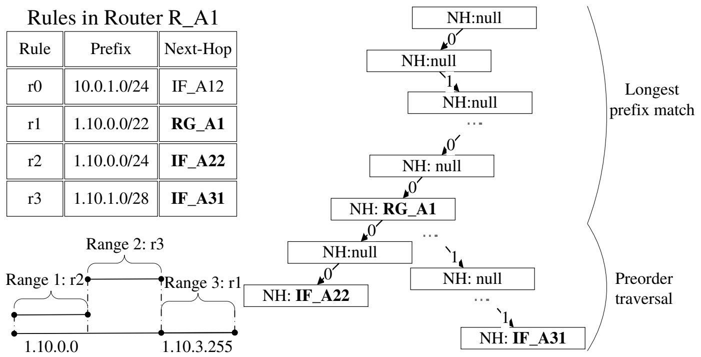


Fig. 5. Example Application of Our Three-Step Verification Approach

B. Data Models

In order to further improve the scalability and response-time of our approach, we investigated prefix matching and packet classification literature. According to our findings, we found out that both X-fast binary tries [16] and radix binary tries [15] fit our purpose. Different type of trie structures have been used in prior works e.g., VeriFlow [21]. Indeed, X-fast binary tries not only allow efficiently storing of all IP addresses with their prefix relationships but also provide fast insertion and searching operations. Furthermore, radix tries are efficiently used to store routing and filtering rules as well as efficiently matching them against packet-headers. In the following, we show how we use them in our approach.

1) Routing and Security Groups: We employ radix tries to store routing and firewall rules and then to perform efficient rule matching against IP prefixes. We use variables for labeling nodes to store information about the rules and their order.


Rules in Router R_A1

Rule	Prefix	Next-Hop
r0	10.0.1.0/24	IF_A12
r1	1.10.0.0/22	**RG_A1**
r2	1.10.0.0/24	**IF_A22**
r3	1.10.1.0/28	**IF_A31**

Fig. 6. An Illustrative Routing Table in Router R_A1 and an Excerpt of the Corresponding Radix Trie

Example 4: Figure 6 illustrates an example of a radix trie (right-side) for an excerpt of the routing rules of router `R_A1` (upper left-side) with the different IP ranges (lower left-side) resulting from matching all rules with the IP prefix of `ExtNet_1` (i.e., 1.10.0.0/22). Edges of the radix trie are labeled with binary values and nodes store different information relevant to matching the bit-strings formed by concatenating all labels of upstream edges starting from the root node [15]. The matching consists in transforming the IP prefix into a bit string (i.e., 0000.0001.0000.1010.00) and using it as a key search to find the corresponding node. The node's variable *NH* stores the matching rule's next hop; for firewall rules (not shown in this example), we use two variables, namely *VAL* for decision of the matching rule (i.e., accept/deny), and *SN* for rules' order. In case of absence of a matching rule, those values are set to null. For instance, the matched node is labeled $NH = RG\_A1$, which corresponds to the next hop specified by rule $r1$ in the routing table and it represents the longest-prefix matched rule.

For routing rules matching with an IP prefix, the common algorithm used by routers for matching a single packet, namely, the longest-prefix match [38], would not be sufficient. Therefore, we only apply the longest-prefix match algorithm to any rule that matches the destination prefix. Then, we apply a pre-order traversal of the sub-trie starting from the node storing the longest-prefix matching rule. The rationale is that other more specific prefixes stored deeper in the radix trie (e.g., for a specific address range) will be needed for a consistent matching result, which would result in splitting the matched IP prefix into ranges, where each range is governed by the appropriate rule, as it will be demonstrated in the following.

Example 5: As depicted in Figure 6, once rule $r1$ is found using the longest prefix match algorithm for the IP prefix 1.10.0.0/22, the preorder traversal algorithm is applied on the sub-trie from the node matching with $r1$. Thus, rules $r2$ and $r3$ are also found to match 1.10.0.0/22. Considering all matching rules, the destination prefix IP is split into three ranges, namely, range 1: 1.10.0.0 $\cdots$ 1.10.0.255, range 2: 1.10.1.0 $\cdots$ 1.10.1.15, and range 3: 1.10.2.0 $\cdots$ 1.10.3.255, respectively governed by r2, r3 and r1.

Note that for matching rules in security groups, we will use the first-match algorithm [39].

2) Prefix-to-Prefix Verification Results Processing: The X-fast binary tries [16] are used (Algorithm 1 in Section IV-C1) to store and progressively compute verification results, per hop, in order to assess isolation between two IP prefixes (see Figure 7). An X-fast trie (denoted by $BTries$) is a binary tree, where each node, including the root, is labeled with the common prefix of the corresponding destination sub-tree. As in radix tries, the left child specifies a 0 bit at the end of the prefix, while the right child specifies a bit-value 1. Each node, including leaves, is labeled with the bit-string from the root to that leaf. We use the leaves to store intermediate and final results as explained in this example. The binary trie's leaves are created and modified progressively by the *prefix-to-prefix* Algorithm 1.

Example 6: Figure 7 illustrates an example of intermediate values of a $BTries$ built for source subnet `SN_A2` and destination `ExtNet_1`. Leaves store the results of matching the radix trie of Figure 6 with destination IP prefix $1.10.0.0/22$, which is actually the root of the X-fast binary trie. Three variables are used at the leaf nodes:

- Variable B stores the boundary of the IP ranges for each leaf. Its value is either L for the lowest bound, H for the highest bound, or LH if a single leaf with a specific IP address (e.g., $1.10.0.2/32$). The leftmost leaf in Figure 7, B is set to L, which means the current leaf is the lowest bound of the IP range `Range 1`. The next leaf, B is set to H to delimit the upper bound of `Range 1`.
- Variable RLB is a two-bit flag that indicates the status of the verification process, where possible values are 00 for no decision yet, 01 for loop found, 10 for blackhole found, or 11 for reachability verified. In the leftmost leaf of the binary trie of Figure 7, RLB is set to 00, which means that the verification is still ongoing for `Range 1` and next hop should be evaluated based on the next variable HR.
- Variable HR is a sequence of triplets (r_id, r_if, src) that stores the history of the visited nodes from source for that IP range, where r_id is a router id, r_if is a router interface and src is the original source node. The last result is appended to the beginning of the sequence and should be used at the next iteration. For more readability, in Figure 7, we only show the two first items of the triplet from the last outcome (next hop) of routing rules matching in `R_A1`.

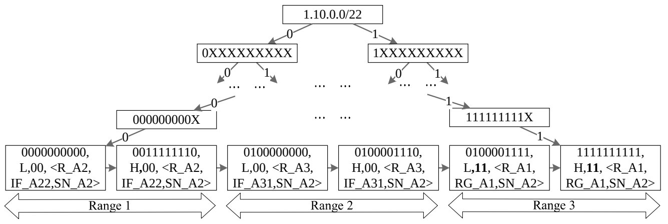

1.10.0.0/22
0
1
0XXXXXXXXX
1XXXXXXXXX
000000000X
111111111X
0000000000, L,00, <R_A2, IF_A22,SN_A2>
0011111110, H,00, <R_A2, IF_A22,SN_A2>
0100000000, L,00, <R_A3, IF_A31,SN_A2>
0100001110, H,00, <R_A3, IF_A31,SN_A2>
0100001111, L,11, <R_A1, RG_A1,SN_A2>
1111111111, H,11, <R_A1, RG_A1,SN_A2>
Range 1
Range 2
Range 3

Fig. 7. X-fast Binary Trie for Subnet SN_A2 and Destination $\mathbf{1.10.0.0/22}$. Leaves Contain Results from Matching Radix Trie in Figure 6 with the Destination

C. Verification

In this section, we present our customized algorithms to perform the three verification steps. The reason that we opt for customized algorithms, instead of existing large-scale graph processing systems (e.g., Pregel [40], BGL [41], and CGMgraph [42]) is that those are mostly designed for general-purpose graph algorithms like finding shortest-path. None of them can easily support network isolation use cases addressed in this paper, in particular, path modifications caused by decision making along the path (e.g. routing, firewalling), or the path transformational operations (e.g., NAT).

Before starting the actual verification, X-fast binary tries are created and initialized for each pair of source and destination IP prefixes using the virtual network model G as it was explained in Section IV-B2. Also, as it was mentioned earlier, both `Step 1` and `Step 2` are parts of the prefix-level verification, where the first step is applied on private IP addresses while the second takes care of the public IP addresses. As a result, we will have $BTries$ for pairs of private IP prefixes of subnets in the same component (verified in `Step 1`) and other $BTries$ for pairs IP prefixes of subnets and external networks (verified in `Step 2.a`) and vice-versa (verified in `Step 2.b`). These two steps will be explained in Section IV-C1. Afterward, VM-level isolation verification takes place at `Step 3`, details of which are in Section IV-C2.

1) Prefix-Level Verification: The function *prefix-to-prefix* (see Algorithm 1) uses the initialized X-fast binary tries $btrie$ to verify prefix-level isolation on each hop between all pairs of source and destination IP prefixes. For a given pair of prefixes, the *prefix-to-prefix* verifies routing rules on a per-hop basis. In all hops between a given pair of prefixes, it uses the same corresponding X-fast binary trie (i.e., having one prefix as source specified in leaves and the other as destination specified in the root of the trie) to update the new results according to the results of matching the rules within the node's radix trie against each IP range. The core of this algorithm is the matching process (explained in Section IV-B1) and copying these results from a temporary trie to the $btrie$. The latter is explained better using the following example.

Example 7: Figure 8 illustrates the process of copying the leaves from the temporary binary trie, which contain the outcome of matching R0 rules with the destination IP prefix, to the the main prefix-to-prefix binary trie within the appropriate IP range (Algorithm 1 line 10). Figure 8 illustrates the modified binary trie after applying a hop per address range verification on the trie of Figure 7 with an excerpt of the rules of `R_A3` (left-side) and `R_A2` (right-side). We compute only the decisions of those routers that are related to `Range 1` and `Range 2`. After matching these tables with the destination address, new leaves are created (e.g., `Range 2` is split into `Range 21` and `Range 22`) with new results, while for others (i.e., `Range 1`) only the result is updated in the binary trie, as follows:

- At `R_A2`, no routing rule was matched, thus indicating a black hole (RLB is 10) for range 1.
- At `R_A3`, matching the destination prefix with the corresponding rules results in two matching rules (i.e., `r31` and `r32`), which partitions `Range 2` into two sub-ranges. `Range 21` is handled by `r31`, which leads to a loop ($RLB = 01$) that can be detected by consulting the variable HR. Packets belonging to `Range 22` are handled by rule `r32` and they can reach the router gateway `R_A3` (i.e., $RLB = 11$).

Algorithm 1 takes as input the binary trie identifier then updates the trie progressively by creating new leaves and modifying others using per-hop results. At each iteration, it traverses the leaves of the trie and, for each IP range, it matches the radix trie corresponding to the networking element specified for that range with the destination IP prefix using algorithms in Section IV-B1. The algorithm terminates if a loop or a blackhole is found, or reachability is verified for all ranges. It uses a temporary trie $TempBTrie$, which contains the result of matching the radix trie of the current router with the destination IP prefix located at the root of the binary trie as discussed in Section IV-B1. This temporary trie is generated once, but can be re-used, particularly, for the verification of other IP prefixes as source (e.g., `SN_A2` and `SN_A3` in Figure 3) with the same destination (e.g., `ExtNet_1`) and the same router (e.g., `R_A1`). Function $searchTries$ finds, if any, the temporary trie corresponding to the specific router and destination IP range. Function $Copy$ is used to update the main binary trie $btrie$ with the results stored in the temporary binary trie $TempBTrie$ for each specific range as discussed in Example 7.

2) VM-Level Isolation Verification: Prefix-level results computed in Section IV-C1 are used to determine subnets that are not isolated. For those subnets, we need to perform a VM-level isolation verification by checking for each pair of VMs their corresponding security groups using both private and public IP addresses. Algorithm 2 describes the *VM-to-VM* procedure in which function $Route\text{-}Lookup$ checks whether there exists a forwarding path between any two VM ports, whereas the $VerifySecGroups$ function verifies security groups of these VMs.

The VM-to-VM route lookup is to determine whether these VMs belong to reachable subnets by searching in the relevant binary tries leaves using the IP addresses of these VMs. This will determine the leaves with boundaries H and L corresponding to the IP ranges containing VMs' IPs and verifying the value of the flag RLB. This is explained in the following example.

Algorithm 1 *prefix-to-prefix*($btrie$)

```
1: Input/Output: $btrie$
2: counter=0
3: for each range $[L, H]$ in $btrie.leafs$ with $RLB = 00$ do
4:     $router = get(HR, r\_id)$
5:     $dst = getroot(btrie)$
6:     if $searchTries(dst, router) = false$ then
7:         $TempBTrie = Match(RadixTrie(router), dst)$
8:     else
9:         $TempBTrie = getBTrie(dst, router)$
10:    $Copy(btrie, TempBTrie, [L, H])$
11:    counter = counter + 1
12: if $counter \neq 0$ then
13:    prefix-to-prefix($btrie$)
```

Algorithm 2 *VM-to-VM*(VM_{src}, VM_{dest})

```
1: Triepub = getBTrie($VM_{dst}.publicIP.CIDR,\ VM_{src}.\ subnet\_id$)
2: Triepriv = getBTrie($VM_{dst}.privateIP.CIDR,\ router\_id$)
3: $routable = Route\text{-}Lookup(Triepub, Triepriv)$
4: if $routable = true$ then
5:     $VerifySecGroups(VM_{src}, VM_{dest})$
```

Example 8: Consider the case of `VM_A1` and `VM_B2` from our running example shown in Figure 2. Route lookup for this pair is achieved by searching for the two X-fast binary tries denoted by $Triepub$ and $Triepriv$, respectively, in Algorithm 1. The $Triepub$ and $Triepriv$ tries contain the routing results respectively, for the pair (`SN_A2`, `ext_net`) and (`ext_net`, `SN_B2`). Using the public IP of `VM_B2` (i.e., 1.10.1.12, which is within the prefix of ExtNet_1), and the private IP of `VM_A1` (i.e., 10.0.0.12, which is within the prefix of subnet SN_A2), the corresponding binary trie $Triepub$ is shown in Figure 8. By searching the ***Triepub*** (see Figure 8) using the public IP of `VM_B2`, one can find that it falls into `Range 22`. The value of RLB for this range is 11, which indicates the existence of a route from SN_A2 to $ExtNet\_1$. Similarly, $TriePriv$ can be identified using the public IP of `VM_B2`, which is attached to router `R_B1`, and its private IP (i.e., 19.0.0.30). Searching in $Triepriv$ (not shown for the lack of space) for the RLB for using the private IP of `VM_B2` allows concluding on the existence of a route between `VM_A1` and `VM_B2`. More precisely, if RLB in these boundary leaves of both $Triepriv$ and $Triepub$ is equal to 11, we say that a forwarding path exists between these VMs.

At this stage, once a path is found between the subnets of the pair of VMs, we then verify both security groups associated with these VMs. According to the type of communication, either private or public IP will be used. For each VM within a source subnet, we use its egress security group radix trie and perform a first-match with the public or private IP of the destination VM. Then, we use the ingress security group rules of the destination VM and perform a first-match with the public or private IP of the VM source. If both results indicate matching rules with the *accept* decisions, then the pair of VMs can be concluded to be reachable using their public or private IP addresses.

3) Complexity Analysis: Let S be the number of subnets, R be the number of routers between two prefixes (i.e., number

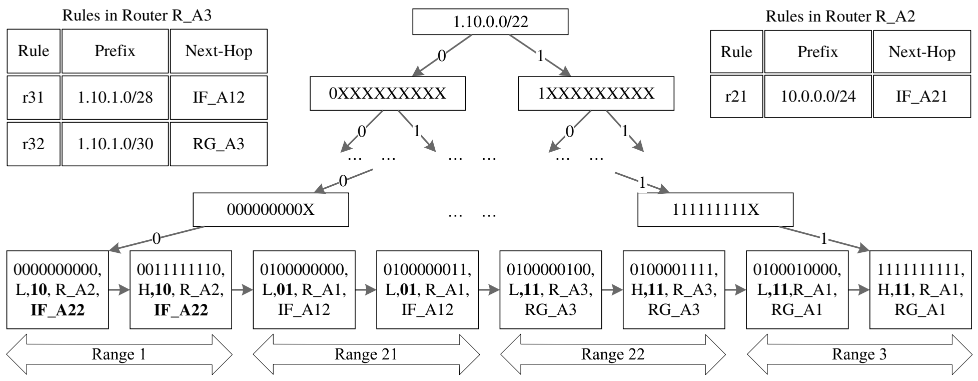

Rules in Router R_A3
Rule | Prefix | Next-Hop
r31 | 1.10.1.0/28 | IF_A12
r32 | 1.10.1.0/30 | RG_A3
1.10.0.0/22
0XXXXXXXXX
1XXXXXXXXX
000000000X
111111111X
Rules in Router R_A2
Rule | Prefix | Next-Hop
r21 | 10.0.0.0/24 | IF_A21
0000000000, L,10, R_A2, IF_A22
0011111110, H,10, R_A2, IF_A22
0100000000, L,01, R_A1, IF_A12
0100000011, L,01, R_A1, IF_A12
0100000100, L,11, R_A3, RG_A3
0100001111, H,11, R_A3, RG_A3
0100010000, L,11,R_A1, RG_A1
1111111111, H,11, R_A1, RG_A1
Range 1
Range 21
Range 22
Range 3

Fig. 8. Updated Binary Trie of Figure 7 Based on Matched Rules in Routers R_A2 and R_A3

of hops), L be the length of keys (whose maximum value is 32 for an exact IP address), M be the number of VMs, and Nex be the number of external networks. Complexities related to the data structure manipulation are known to be $O(L)$ for insert operation in X-fast binary tries, $O(Log(L))$ for search operations in X-fast binary tries, and $O(L)$ for radix trie matching per router.

In `Step 1` and `Step 2`, the complexity of prefix-to-prefix reachability verification (Algorithm 1) is $O((S^2 + 2 \times S \times Nex) \times R \times K \times (L + log(L)))$, where K represents the number of operations performed over the data structures for each routing node. This can be approximated to O(S^2) for large data centers where the number of subnets is larger than the number of external networks ($Nex \ll S$) and the number of hops is usually limited for delay optimization ($R \ll S$), with L and K being constants. In `Step 3`, the complexity of VM-level verification (Algorithm 2) is $O(2 * (L + Log(L)) * M^2)$ and can be approximated to $O(M^2)$.

We thus obtain an overall complexity of $O(S^2 + M^2)$. However, this only provides a theoretical upper bound, which typically will not be reached in practice. In general, depending on the communication patterns in multi-tenant clouds, the number of interconnected subnets is usually smaller than S as traffic isolation is the predominant required property in such environments. For example, it has been reported in [43] that inter-tenant traffic varies between 10% and 35% only. Thus, if we denote by M' the number of VMs belonging to connected subnets, it is safe to claim the practical complexity for our solution would be $O(S^2 + M'^2)$, where $M' \ll M$.

4) Correctness: According to our model, verifying isolation means checking whether there exists any layer 3 communication path between any pair of VMs according to tenants' policies. Therefore, proving the correctness of our approach boils down to proving that our algorithm visits all paths and returns the desired isolation result for each of them. In a typical cloud environment at network virtual layer 3, private IP addresses are used for communications inside the same network (component), whereas public addresses are used for communications between VMs belonging to different networks, and with networks outside the cloud. As we are considering both private and public IP addresses (with NAT mechanism) to investigate the whole symbolic IP packet address space, our approach explores all IP forwarding paths by iteratively applying, on each path, relevant forwarding and filtering functions (using corresponding matching algorithms) of each encountered node in the virtual network connectivity graph for each packet.

Referring to Figure 4, Step 1 and Step 2 explore disjoint prefix-level IP address spaces (private addresses spaces vs public addresses space). Thus, the two steps do not have any side effects on one another. The results of these two steps are the pairs of subnets that can reach each other (R) and those that cannot (U). As we use well-known packet header matching algorithms to find reachable paths, the sets U and R should contain the correct pairs with respect to their reachability status. The third step relies on the results of Step 1 and Step 2 and verifies security groups for all pairs of VMs belonging only to the set of pairs of reachable subnets in R. In this step as well, we rely on state-of-the-art first-match algorithm applied on firewall rules at each VM-side against the header of the symbolic packet. Therefore, the correctness of our approach follows from the correctness of those well-established algorithms in a straightforward manner.

5) Incremental Verification: The dynamic nature of cloud leads to frequent changes in the configurations of virtual networks. The verification result may be invalidated even after a single change, such as the deletion of a security group rule from a group of VMs, or the addition of a routing rule to a router. However, verifying the cloud-wide network isolation again after each such event is obviously costly and unnecessary.

To cope with the effect of each event at run-time, TenantGuard adopts an incremental event-driven verification approach. This approach first identifies the set of events that potentially impact the isolation results. Then, the impact of each such event is identified. Finally, only those parts of the verification that are affected by the event will be re-evaluated. Table II lists an excerpt of events that may require updating the verification results along with their impact. Note that G should be updated for all these events and the symbol $*$ in the table indicates the network elements impacted by the event.

To illustrate how such events may be handled via incremental verification, Algorithm 3 sketches the steps for partially updating the verification results upon deleting a security group rule and upon adding a new routing rule, respectively, as explained in the following (detailed algorithms for other events are omitted due to space limitations).

- **Creating a VM**: The creation of a new VM (denoted as VM*) does not affect the verification process unless it gets connected to one or more subnets through virtual ports, which naturally leads to the update of our graph model by creating the corresponding virtual port nodes. Furthermore, when the VM is first created, it is attached to

Event	Verification Tasks
Creating a VM*	• Invoke *VM-to-VM* for VM* once as source and once as destination
Deleting a VM*	• Remove the results related to *VM-to-VM* for VM*
Creating a subnet SN*	• Initialize a radix trie for the host routes • Create new prefix-to-prefix binary tries where SN* is source or destination • Invoke *prefix-to-prefix* for subnets in the C* related to SN* • Invoke *prefix-to-prefix* (Step 2.a and Step 2.b) for SN*
Deleting a subnet SN*	• Delete the prefix-to-prefix tries where SN* is source or destination • Update VM-level isolation for all VMs having private IPs within the prefix of SN* either as source or destination
Creating a router R*	• Initialize the corresponding radix trie
Deleting a router R*	• Recalculate all the prefix-to-prefix tries for the component C* related to R* • Partially perform VM-level isolation for all VMs belonging to C* considered as source and as destination

TABLE II. AN EXCERPT OF EVENTS AND THEIR CORRESPONDING INCREMENTAL VERIFICATION TASKS. THE SYMBOL * INDICATES THE NETWORK ELEMENTS THAT ARE IMPACTED BY THE EVENT

the default security group. At this level, the results of step 1 and step 2 of our methodology remain unchanged and the verification update is confined to step 3 by invoking the function VM-to-VM.

- **Deleting a VM**: After deleting a VM, the graph model is updated by removing the associated virtual ports, then the last result is updated by removing all VM pairs where the deleted VM appears either as a source or as a destination.

- **Creating a subnet**: When a subnet (denoted as SN*) is newly created, it is specified with a gateway, which is a router interface, and an IP prefix. Our graph model is updated with a new subnet node with an edge to the gateway interface and the corresponding radix tree is initialized. This event will create new prefix-to-prefix binary tries for which SN* is either a source or a destination. This would result in re-calculating step 1 for C*, the maximally connected component SN* belongs to, then step 2-a and step 2-b for SN*. As long as no VM has been attached to SN*, the VM-to-VM reachability verification (step 3) does not require updates.

- **Deleting a subnet**: The deletion of a subnet (denoted as SN*) will lead to deleting the prefix-to-prefix tries where SN* appears either as a source or as a destination. This will obviously reduce the number of possible forwarding paths. As such, VM-to-VM reachability also needs to be updated accordingly for all VMs having their private IPs within the prefix of SN* either as a source or as a destination.

- **Creating a router**: The event of adding a router (denoted as R*) would result only in adding a router node in the graph model and initializing the corresponding radix tree. The verification result is affected when the router's interfaces are connected either to the tenant's network or to the external network, and the routing rules are added.

- **Deleting a router**: Deleting a router R* requires recalculating all the Btries for the component C* the router belongs to. VM-to-VM reachability analysis should also be partially performed in this situation for all VMs belonging to C* either considered as a source or as a destination.

- **Deleting a security group rule**: Whenever an ingress (or egress) rule is deleted from a security group, the action is propagated into all VMs, denoted as VMs*, this security group is attached to. Consequently, the corresponding ingress (or egress) radix trie is updated accordingly. Let VM* be a member of VMs*. For the deletion of an egress (resp. ingress) rule, security groups verification result is updated for all pairs where VM* appears as a source VM (resp. destination VM).

- **Adding a routing rule**: Whenever a new routing rule is added to a router, denoted as R*, belonging to a component C*, this would result in updating the corresponding radix trie with the decision of the newly inserted rule. Then, for each prefix-to-prefix binary trie built for subnets belonging to the component C*, the variable HR of the binary trie (holding the history of visited nodes) is consulted. If the ID of R* appears in the history of traversed nodes, then the corresponding binary trie needs to be updated. Then VM-level isolation needs to be checked for couples of VMs if the source and/or destination belong to C*. Routing rules and host routes deletion and addition events are handled similarly.

Algorithm 3 *Rules addition/deletion*

```
On the creation/deletion of a security group rule r* for a set of VMs* do:
update RadixTrie(r*)
for each VM* in VMs* do
    if r* is an egress rule then
        for each pair $(VM_{src}, VM_{dst})$ where $(VM_{src} = VM^{*}$ do
            $VerifySecGroups(VM^{*}, VM_{dest})$
    if r* is an ingress rule then
        for each pair $(VM_{src}, VM_{dst})$ where $(VM_{dst} = VM^{*}$ do
            $VerifySecGroups(VM_{src}, VM^{*})$
On the creation/deletion of a routing rule r* at router R* belonging to C* do:
update RadixTrie(r*)
for each $prefix\text{-}to\text{-}prefix$ binary trie $btrie$ built for subnets of $C^{*}$ do
    if $R^{*} isinbtrie.leaves.HR$ then
        $prefix\text{-}to\text{-}prefix(btrie)$
for each pair $(VM_{src}, VM_{dst})$ where $VM_{src} inC^{*}$ and/or $VM_{dst} inC^{*}$ do
    $VM\text{-}to\text{-}VM(VM_{src}, VM_{dst})$
```

To facilitate the verification update, we leverage caches that store intermediary and previous prefix-level isolation results, such as X-fast binary tries. We also utilize the radix tries, which store the routing rules and the security groups.

As discussed in Section IV-C3, the complexity of Algorithm 3 (basically updating the Radix tree) is constant because it is linear in the length of a key, which is bounded by 32. In contrast, the complexity of a full verification is $O(M^2)$ where M is the number of VMs, can be as large as millions for a real cloud. Therefore, the overhead of our incremental verification is negligible in comparison to a full verification.

V. APPLICATION TO OPENSTACK

We have implemented the proposed system design as a prototype system based on OpenStack [12]. In this section, we briefly discuss implementation details about data collection, preprocessing, and parallel verification. In OpenStack, VMs are managed by the compute service `Nova`, while networking service `Neutron` manages virtual networking resources in the cloud. Data related to these services is stored in databases containing over one hundred tables.

Data Collection and Preprocessing. TenantGuard allows both on-demand and on regular basis incremental auditing. To build a snapshot of the virtual networking infrastructure for auditing, we collect data from OpenStack databases. Additionally, we leverage the notification service from PyCADF [44] and Ceilometer [12] services to intercept operational events that result in a configuration change. Thus, our data collection module starts by collecting an initial snapshot of the virtual networking infrastructure. Then, at each detected event, the changed configuration is gathered and the snapshot of the virtual networking infrastructure is updated to enable incremental verification.

Once the data is collected, we perform several preprocessing steps, such as building and initializing different data structures to be used in the verification step. For instance, from the list of all subnets, routers, and gateways of all tenants, we correlate the information to determine which subnets can actually communicate through public IPs. We also determine the lists of subnets involved in the prefix-level verification using public IP prefixes, and subnets per maximally connected subgraphs as explained in Section IV-C1. Additionally, we filter all ‘orphan’ subnets, as they are not connected to any other subnets or external networks.

In OpenStack, VMs and virtual networking resources are respectively managed by Nova and Neutron services. The corresponding configuration data is then stored in Nova and Neutron databases. Therefore, we mainly used SQL queries to retrieve data for different tables in those databases. For instance, VM ports, router interfaces, router gateways and other virtual ports are collected from table ports in Neutron database. Therein, we use both device owner and device id fields to infer the type and affiliation relationship between the virtual ports and their corresponding devices. The packet filtering and forwarding rules are stored in neutron.routerrules, subnetroutes, and securitygrouprules tables, where rules are represented by destination-nexthop data pairs.

Parallelization of Reachability Verification. In addition to the single machine-based implementation, we have also extended TenantGuard to a parallel environment. The parallelization is based on building groups of prefixes such that there is no common path in the graph. This allows us to cache the temporary binary tries to store results for routers matching, which can be reused in other paths. Thus, we divide the list of all prefixes into groups of prefixes that would be used as destination prefixes and we create the same number of threads as the groups, where each thread considers all possible prefixes as sources.

The analysis controller is responsible for data collection, graph construction, verification tasks scheduling and distribution. The controller obtains the topological view of the compute worker cluster, its computation capacity and metrics, such as the number of cores with CPU loads. Based on such profiles, the task scheduler dynamically divides the verification computation into Java runnables, which will be distributed and executed individually across the worker clusters through data streaming in such a way that all the tasks are performed in memory and no disk IO is involved. To avoid interference, the divided runnable is not executed at the controller. The compute worker cluster consists of nodes for performing tasks assigned by the controller. The nodes discover each other automatically through the configuration in the same LAN. The result could be returned directly to the caller, or written into the data cache cluster in such a way that the data can be in-memory distributed among the nodes. The latter is especially useful when the size of data exceeds the capacity of single-machine memory.

Integration to OpenStack Congress. We further integrate TenantGuard into OpenStack Congress service [17]. Congress implements policy as a service in OpenStack in order to provide governance and compliance for dynamic infrastructures. Congress can integrate third party verification tools using a data source driver mechanism [17]. Using Congress policy language that is based on Datalog, we define several tenant specific security policies. We then use TenantGuard to detect network isolation breaches between multiple tenants. TenantGuard’s results are in turn provided as input for Congress to be asserted by the policy engine. This allows integrating compliance status for some policies whose verification is not supported by Congress (e.g., reachability verification as mentioned in Section II). TenantGuard can successfully verify VM reachability results against security policies defined inside the same tenant and among different tenants. TenantGuard can also detect breaches to network isolation. For example, we test an attack in which, through unauthorized access to the OpenStack management interface, the attacker authorizes some malicious VMs to have access to the virtual networks from other tenants. TenantGuard can successfully detect all such injected security breaches providing the list of rules in the routers that caused the breach.

VI. Experiments

This section presents experimental results for performance evaluation of TenantGuard on a single machine, on Amazon EC2 [45] and using data collected from a real cloud. We also perform a quantitative comparison with our baseline algorithm and with NoD [27], which is the closest work to ours (as detailed in Section II, most of the other works are either designed for physical networks and not suitable for large scale virtual networks, or they do not support the verification of all-pair reachability at the VM-level as targeted by our solution). Note that, in our experiments, the baseline algorithm is not brute force but already an optimized algorithm that uses efficient data structures, mainly radix tries (although it lacks the other optimization mechanisms of our final solution, e.g., the three-step prefix-to-prefix approach detailed in Section IV).

A. Experimental Settings

Our test cloud is based on OpenStack version Kilo with Neutron network driver, implemented by ML2 OpenVSwitch and L3 agent plugins, which are popular networking deployments [12]. There are one controller node integrated with networking service, and up to 80 compute nodes. Tenants’ VMs are initiated from the Tiny CirrOS image [12], separated by VLAN inside the compute nodes, while VxLAN tunnels are used for the VM communications across the compute nodes.

We generate two series of datasets (i.e., SNET and LNET) for the evaluation. The SNET datasets represent small to

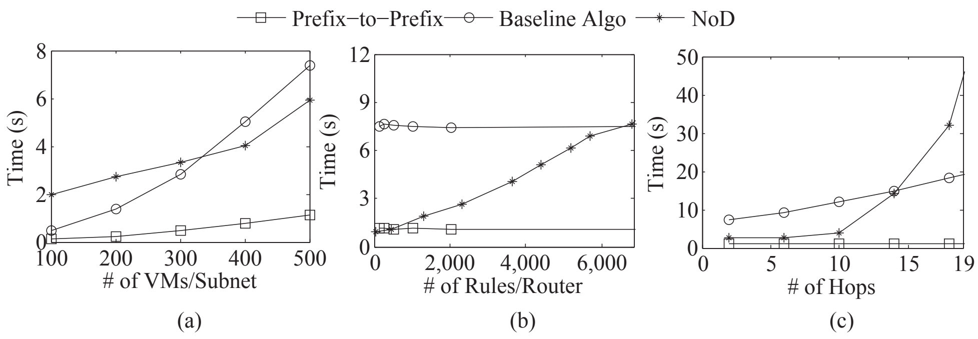


Fig. 9. Performance Comparison by Varying the # of (a) VMs per Subnet, (b) Routing Rules, and (c) Hops, while Fixing the # of Subnets to 6[2]

DataSet	VMs	Routers	Subnets	Reachable Paths
DS1	4362	300	525	$>$ 5.67 million
DS2	10168	600	1288	$>$ 29.2 million
DS3	14414	800	1828	$>$ 57.0 million
DS4	20207	1000	2580	$>$ 109 million
DS5	25246	1200	3210	$>$ 168 million

TABLE III. LNET DATASET DESCRIPTION

medium virtual networks containing six subnets, while we vary different factors such as the number of VMs per subnet, the number of rules per router, and the number of hops between subnets, to examine corresponding characteristics of our algorithms, and have been built using OpenStack and specifically using Horizon. We then cloned the SNET virtual infrastructure environments to obtain different tenants and thus LNET datasets, which represent large networks, where each virtual network is organized in a three-tier structure where the first-tier router is connected to the external network, while the others use extra routes to forward packets between each other, so in essence they are synthetic. Security rules are generated in the same logic behind the deployment of two-tier applications in the cloud. For a given tenant, one group of VMs can only communicate with each other but not with outside (or other tenants) networks, while another group is open to be reached from anywhere. Up to 25,246 VMs are created in the test cloud, with 1,200 virtual routers, 3,210 subnets, and over 43,000 allocated IP addresses. As a reference, according to a recent report [12], 94% of interrogated OpenStack deployments have less than 10,000 IPs. Therefore, we consider the scale of our largest dataset is a representative of large size clouds. The five datasets in *LNET* are described in Table III. We use open-source Apache Ignite [46] as the parallel computation platform, which can distribute the workload in real-time across hundreds of servers. On the other hand, all datasets both in SNET and LNET for NoD are generated synthatically using the provided generator.[3]

B. Results

We evaluate the performance of our approaches and the effect of various factors on the performance.

1) SNET Results: This set of experiments is to test how network structure and configuration influence the performance of our system. All tests using *SNET* datasets are conducted with a Linux PC having 2 Intel i7 2.8GHz CPUs and 2GB memory. Note that, for SNET datasets the verification time for NoD is measured for 1 to 5 pairs, and for TenantGuard the time is measured for all-pair. As shown in Figure 9(a), when the number of VMs per subnet is increased from 100 to 500, the prefix-level isolation verification time increases much slower than the baseline algorithm (defined in Section IV-A) and NoD. The reason behind these results is, as illustrated in complexity analysis in section IV, the prefix-to-prefix algorithm that reduces the complexity to $O(R+N^2)$, in contrast to $O(R*N^2)$ in the baseline algorithm, where R is the number of hops and N is the number of VMs; when N increases, the complexity $O(R*N^2)$ increases much faster than $O(R+N^2)$. On the other hand, as the number of pairs is one of the major factors for NoD verification time, we observe increase in the verification time while increasing the number of pairs from 1 to 5. As shown in Figure 9(b), when the number of routing rules per router increases exponentially, the verification time for TenantGuard and the baseline algorithm remain relatively stable due to using radix trie and X-fast binary trie, both of which have constant searching time; However, the baseline algorithm takes longer time due to the higher number of pairs to be verified. On the other hand, as NoD is designed for a large number of rules instead of large number of pairs, we increase the number of pairs from 1 to 5 while keeping the number of rules similar to the setting of TenantGuard.

Additionally, as the number of hops increases the complexity of the verification (it corresponds to the number of virtual routers on a communication path), we vary the number of hops between VMs. We investigate the average number of hop usually encountered in real life systems (e.g., Internet) and according to [47] and [48], the average number of hops varies between 12 to 19; hence, we vary the number of hops between 2 to 19. Figure 9(c) shows that the prefix-to-prefix verification experiences negligible changes. In contrast, a fourfold increase in the overhead is observed with the baseline algorithm. Whereas, the verification time for NoD increases exponentially specially after 14 hops, as their algorithms are not optimized for higher number of hops.

2) LNET Results Using Amazon EC2: **Single-Machine Mode.** The *LNET* datasets are used to examine the scalability of our system for large virtual networks. Hence, factors examined in the *SNET* dataset are kept invariant for each subnet, and the number of tenant's subnets is varied as shown in Table III. There are two modes for *LNET* tests: single-machine mode and parallel mode. Single-machine tests are conducted on one EC2 C4.large instance at AWS EC2 with 2 vCPUs and 3.75 GB memory. We measure NoD performance only for the single machine tests, as NoD implementation does not support parallelization. The data collection and initialization steps are performed on a single machine in both modes.

[2]Note that for NoD, we vary the number of pairs from 1 to 5 through the X axis, and for TenantGuard, we consider all possible pairs of VMs as the X axis depicts the number of VMs.

[3]Available at: http://web.ist.utl.pt/nuno.lopes/netverif

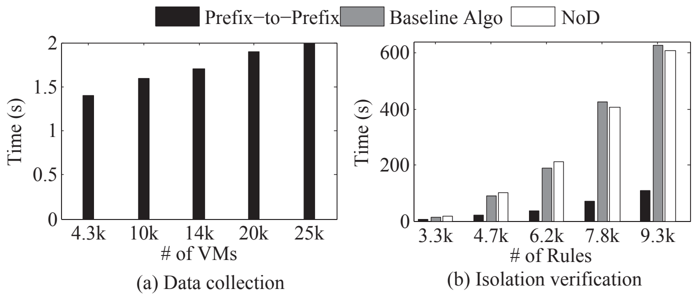


Fig. 10. Performance Comparison in the Single-Machine Mode with the LNET Datasets Described in Table III. (a) Showing Data Collection Time, and (b) Showing Verification Time

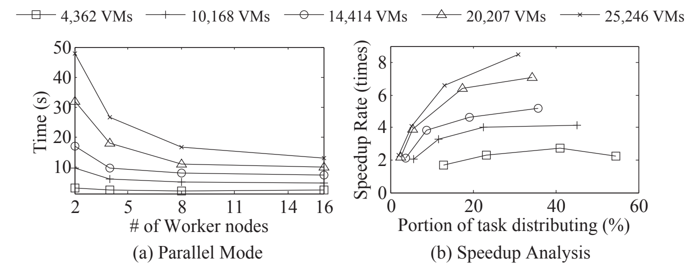


Fig. 11. The Performance Improvement of Parallel Computation with LNET Data Described in Table III. (a) Verification Time while Varying the Number of Worker Nodes in Amazon EC2 for Different Datasets, and (b) Speedup Analysis over the Number of VMs Using 16 Worker Nodes

As shown in Figure 10(a), data collection and processing time varies between 1.5 to 2 seconds, including retrieving data from the database, initializing radix tries for routers and security groups, etc., which shows that the collection time is not the prominent part of the execution time. Meanwhile, Figure 10(b) compares the verification time between TenantGuard, baseline algorithm and NoD. When the number of routing rules increases along with subnets and VMs, the prefix-to-prefix algorithm is more efficient than NoD and the baseline algorithm (e.g., TenantGuard performs 82% faster than NoD for the largest dataset). NoD (while varying the number of pairs from 20 to 200 through the X axis) and the baseline algorithm show almost similar response time. For 9,300 routing rules with 25,246 VMs in 3,250 subnets, it takes 108 seconds using the prefix-to-prefix algorithm, 605 seconds for NoD (for 200 pairs) and around 628 seconds for the baseline algorithm. Note that TenantGuard verifies in total over 168 millions VM pairs.

We report our extended experiment results to further validate the scalability of TenantGuard in Table IV. In this set of experiment, we increase the number of routing rules to 850k, and the number of VMs to 100k to compare the reported results in Plotkin et al. [11]. For the first part, we compare TenantGuard with NoD for the 850k routing rules with other parameters as our DS5 datasets, and observe that TenantGuard completes the all-pair reachability verification in 100.14s, which is significantly faster than NoD (5.5 days). In the second part, we generate a completely new datasets with 850k routing rules and 100k VMs, and observe that the all-pair reachability verification takes less than 18 minutes for TenantGuard, whereas Plotkin et al. [11] needs about 2 hours.

Datasets	850k rules	850k rules and 100k VMs
NoD [27]	475,200 [11]	-
Plotkin et al. [11]	-	7,200 [11]
TenantGuard	100.14	1,055.88

TABLE IV. COMPARING THE PERFORMANCE (IN SECONDS) BETWEEN EXISTING WORKS AND TENANTGUARD TO VERIFY ALL-PAIR REACHABILITY

Parallel Verification Test. Although our approach already demonstrates significant performance improvements over NoD and the baseline algorithm, the results are still based on a single machine. In real clouds, with large deployments (10Ks of active VMs), there is need for verifying very large virtual networks. Therefore, we extend our approach to achieve parallel verification, where the isolation verification is distributed among the nodes of worker cluster, except the data collection and initialization run on a single node. This parallel implementation provides larger memory capacity to our approach, and results in much shorter verification times. For the parallel mode, one EC2 C4.xlarge instance with 4 vCPUs is configured as the controller, and up to 16 instances of the same type with a compute worker cluster, while node discovery and communications are established by their internal IPs.

Figure 11(a) shows the performance of parallel verification using 2 to 16 worker nodes. Clearly, for each dataset, by increasing the number of worker nodes, in contrast to the result of the single machine mode, the overheads decrease significantly. For example, in contrast to 108 seconds in the single machine mode, it only takes approximately 13 seconds in the parallel mode with 16 workers, while over 160 millions of paths are verified as reachable.

In Figure 11(b), in order to show the scalability of our approach while increasing the virtual network size, we examine the relationship between the cluster size and speedup gain. The parallel execution time can be divided into two parts: task distribution time to send input data from the controller to different workers, namely T_d, and execution time on those nodes (we ignore the result generation time due to the small size of result data). We note that, even if the tasks could be divided evenly, which is unlikely the case in practice, the tasks could still arrive at worker nodes at different times. As a result, some of those tasks may start significantly later than others due to networking delay, while the overall performance is always decided by the slower runners. As T_d becomes larger, it becomes more predominant in the overall execution time. However, due to the lack of knowledge on task execution sequence in the synchronous mode, we cannot accurately measure the distribution time. Additionally, there will always be some tasks which begin later than the other tasks. In order to minimize this impact and to start tasks at roughly the same time, we use an asynchronous task distribution technique. In Figure 11(b), the x-axis represents the ratio T_d/T, while T is the overall verification time. In addition, the speedup ratio (R_s) is the performance ratio between sequential and parallel programs, represented by y-axis. With the number of worker nodes increasing, T_d rises as expected because more data and code need to be transferred among cluster nodes. When it becomes more dominant, the speedup rate increases more gradually. For the smallest dataset, R_s decreases when the number of workers ranges from 8 to 16. The T_d/T ratio can be used to decide the optimal data size in each node.

Our experiment results show that even a small number (i.e., 16) of working nodes can handle large-scale verification (i.e., 168 millions of VM pairs); recalling that real world clouds have the size of 100,000 users and maximum 1,000 VMs for each user. Also, our speed-up analysis (Figure 11(b)) illustrates that after 8 nodes the speedup goes down. Therefore, we restrict the number of working nodes to 16. The result of incremental verification is not reported, as our discussion

Routing/Filtering	OpenStack [12]	Amazon EC2-VPC [49]	Google GCE [50]	Microsoft Azure [51]	VMware vCD [52]	TenantGuard support
Intra-tenant routing	Host routes, routers	Routing tables	Routes	System and user-defined routes	Distributed logical routers	Yes/ TenantGuard forwarding and filtering function
Inter-tenant routing	Routers, external gateways	Internet gateway/VPC peering	Internet gateway	System route to Internet	Edge gateway	Yes/ TenantGuard forwarding and filtering function
L3 filtering	Security groups	Security groups	Firewall rules	Network security groups	Edge firewall service	Yes/ TenantGuard forwarding and filtering function

TABLE V. ROUTING AND FILTERING IN DIFFERENT CLOUD PLATFORMS AND HOW THEY ARE SUPPORTED IN TENANTGUARD

in Section IV-C5 shows that the overhead of the incremental verification is negligible in comparison to a full verification.

3) Experiment with Real Cloud: We further test TenantGuard using data collected from a real community cloud hosted at one of the largest telecommunications vendors. The main objective is to evaluate the real world applicability of TenantGuard (this dataset is not suitable for performance evaluation due to the relatively small scale of the cloud). All tests are performed in a single machine using the collected dataset without any modification. The tested cloud consists of only nine routers and 10 subnets. Initially, the TenantGuard verification process fails due to a minor incompatibility issue between the OpenStack version used in our lab (Kilo) and an earlier version used inside the real cloud (Juno). From OpenStack Juno to Kilo, two new fields are added to the neutron.networks table, namely, 'mtu' int(11) and 'vlan_transparent' tinyint(1). This difference between the two versions has prevented TenantGuard to execute SQL queries against table neutron.networks due to the missing 'mtu' field. After addressing this issue by altering the neutron.networks table, TenantGuard successfully completes the requested verification in several milliseconds.

VII. DISCUSSION

In this section, we provide the required effort to adopt TenantGuard in other cloud platforms e.g., Amazon, Google, VMware. Additionally, we discuss existing methods to build a chain of trust to preserve the integrity of the collected data.

Adapting TenantGuard in other cloud platforms. we review packet routing and filtering in different cloud platforms and show the applicability of TenantGuard. Table V shows how routing and filtering are implemented in OpenStack, Amazon AWS EC2-VPC (Virtual Private Cloud) [49], Google Compute Engine (GCE) [50], Microsoft Azure [51], and VMware vCloud Director (vCD) [52]. Similar to OpenStack, all other platforms allow tenants to create private networks and to create routing rules to govern communication between them. Those rules are captured by the forwarding and filtering function fd_G in our model. VMs attached to those private networks can have private IPs and public IPs respectively for intra-tenant and inter-tenant communication. In the case of inter-tenant communication, gateways are endowed with NAT services in order to manage mapping between private and public IP addresses. NAT rules are captured in our model by the function fd_G. Internet gateways in EC2-VPC, system route to the Internet in Azure, and edge gateways in vCD can be represented in our model by the component *v_router_gw*. Exceptionally, in EC2 VPC, the VPC peering routing can be employed to enable private IP connections across tenants' virtual networks with tenants' agreement. To support this feature, the definition of fd_G will need to be extended. Security groups in OpenStack and EC2, firewall rules in ECG, network security groups in Azure, and edge firewall services in vCD are set up to filter VMs' outbound/inbound packets. Those filtering rules are also supported by TenantGuard via the forwarding and filtering function fd_G.

Preserving integrity of the system. There exist many techniques on trusted auditing to establish a chain of trust, e.g., [53], [54], [55]. Bellare et al. [53] propose a MAC-based approach. They provide the forward integrity by using a chain of keys and erasing previous keys so that any old logs cannot be altered. Crosby et al. [54] also present a tree-based history data structure, which prevents log tampering where the author of the log is untrusted. Apart from tamper prevention, there are some other works to further detect tampering logs. Chong et al. [57] implement the Schneier and Kelsey's secure audit logging protocol with tamper resistant hardware, namely iButton. Furthermore, OpenStack leverages Intel Trusted Execution Technology (TXT) to establish a chain of trust from the embedded TPM chips in the host hardware to critical software components using a standalone attestation server [56].

VIII. CONCLUSION

In this paper, we have proposed a novel and scalable runtime approach to the verification of cloud-wide, VM-level network isolation in large clouds. We presented a new hierarchical model representing virtual networks, and we designed efficient algorithms and data structures to support incremental and parallel verification. As a proof of concept, we integrated our approach into OpenStack and also extended it to a parallel implementation using Apache Ignite. The experiments conducted locally and on Amazon EC2 clearly demonstrated the efficiency and scalability of our solution. For a large data center comprising 25,246 VMs, verification using our approach finished in 13 seconds. The main limitations of this work are as follows. First, since TenantGuard only focuses on the virtual network layer, a future direction is to integrate it with existing tools working at other layers (e.g., verification tools for physical networks, or co-residency and covert channel detection techniques). Second, since TenantGuard relies on cloud infrastructures for input data, how to ensure the integrity of such data (e.g., through trusted computing techniques) is another future direction. Third, TenantGuard assumes the verification results can be safely disclosed to tenants, which may not always be the case, and addressing such privacy issues comprises an interesting future challenge.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers and our shepherd, Vyas Sekar, for their valuable comments. We appreciate Yue Xin's support in the implementation. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant N01566.

REFERENCES

[1] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v 3.0, 2011.

[2] Cloud Security Alliance. Cloud computing top threats in 2016, Feb 2016.

[3] V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle. A survey of network isolation solutions for multi-tenant data centers. *IEEE Communications Surveys Tutorials*, PP(99):1–1, 2016.

[4] SANS Institute, InfoSec Reading Room. An introduction to securing a cloud environment, 2012.

[5] Amazon Web Services. Overview of security processes, June 2016.

[6] ISO Std IEC. ISO 27002:2005. *Information Technology-Security Techniques*, 2005.

[7] ISO Std IEC. ISO 27017. *Information technology- Security techniques (DRAFT)*, 2012.

[8] Cloud Security Alliance. Cloud control matrix CCM v3.0.1, 2014. Available at: https://cloudsecurityalliance.org/research/ccm/.

[9] OpenStack. OpenStack user survey, 2016. Available at: https://www.openstack.org.

[10] RightScale. RightScale 2016 state of the cloud report, 2016. Available at: http://www.rightscale.com.

[11] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *POPL*, 2016.

[12] OpenStack. OpenStack open source cloud computing software. Available at: http://www.openstack.org.

[13] OpenStack. Nova network security group changes are not applied to running instances, 2015. Available at: https://security.openstack.org/ossa/OSSA-2015-021.html, last visited on: May, 2016.

[14] OpenStack. Routers can be cross plugged by other tenants, 2014. Available at: https://security.openstack.org/ossa/OSSA-2014-008.html, last visited on: May, 2016.

[15] J. Corbet. Trees I: Radix tree. Available at: http://lwn.net/Articles/175432/.

[16] D. E. Willard. Log-logarithmic worst-case range queries are possible in space o(n), 1983. Information Processing Letters.

[17] OpenStack. Policy as a Service (Congress). Available at: http://wiki.openstack.org/wiki/Congress.

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

[19] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[20] H. Mai, Ahmed Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[21] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *NSDI*, 2013.

[22] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, Oct 2013.

[23] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI*, 2014.

[24] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, 2016.

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

[26] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.

[27] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI'15*, 2015.

[28] Sören Bleikertz, Carsten Vogel, and Thomas Groß. Cloud Radar: Near real-time detection of security failures in dynamic virtualized infrastructures. In *ACSAC*, 2014.

[29] T. Probst, E. Alata, M. Kaâniche, and V. Nicomette. An approach for the automated analysis of network access controls in cloud computing infrastructures. In *Network and System Security*. 2014.

[30] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[31] Taous Madi, Suryadipta Majumdar, Yushun Wang, Yosr Jarraya, Makan Pourzandi, and Lingyu Wang. Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack. In *CODASPY*, 2016.

[32] Suryadipta Majumdar, Yosr Jarraya, Taous Madi, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Proactive verification of security compliance for clouds through pre-computation: Application to OpenStack. In *ESORICS*, 2016.

[33] S. Bleikertz, T. Groß, M. Schunter, and K. Eriksson. Automated information flow analysis of virtualized infrastructures. In *ESORICS*, 2011.

[34] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. Technical report, Technical Report MSR-TR-2014-102, Microsoft Research, 2014.

[35] OpenStack. Congress documentation release. Available at: https://congress.readthedocs.io/en/latest/.

[36] Robin J. W. Definitions and examples. In *Introduction to Graph Theory, Second Edition*, 1979.

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

[38] Fred Halsall. *Computer Networking and the Internet (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[39] W. R. Cheswick, S. M. Bellovin, and A. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*.

[40] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[41] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *POOSC*, 2, 2005.

[42] Albert Chan and Frank Dehne. CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2003.

[43] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *NSDI*, 2013.

[44] Cloud auditing data federation (CADF). PyCADF: A Python-based CADF library, 2015. Available at: https://pypi.python.org/pypi/pycadf.

[45] Amazon. Amazon EC2- Virtual Server Hosting. Available at: https://aws.amazon.com/ec2.

[46] Ignite. Available at: https://ignite.apache.org.

[47] A. Fei, G. Pei, R. Liu, and L. Zhang. Measurements on delay and hop-count of the internet. In *GLOBECOM*, 1998.

[48] F. Begtasevic and P. V. Mieghem. Measurements of the hopcount in internet. In *PAM*, 2001.

[49] Amazon. Amazon virtual private cloud. Available at: https://aws.amazon.com/vpc.

[50] Google. Google compute engine subnetworks beta. Available at: https://cloud.google.com.

[51] Microsoft. Microsoft Azure virtual network. Available at: https://azure.microsoft.com.

[52] VMware. VMware vCloud Director. Available at: https://www.vmware.com.

[53] M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.

[54] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, 2009.

[55] Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions on Storage (TOS)*, 5(1):2, 2009.

[56] OpenStack. Security hardening, 2016. Available at: http://docs.openstack.org/admin-guide/compute-security.html.

[57] Cheun Ngen Chong, Zhonghong Peng, and Pieter H Hartel. Secure audit logging with tamper-resistant hardware. In *Security and Privacy in the Age of Uncertainty*. Springer, 2003.