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

Yu Shun Wang

A Thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Information Systems Security) at

Concordia University

Montréal, Québec, Canada

May 2017

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Yu Shun Wang**
Entitled:	**TenantGuard: Scalable Runtime Verification of Cloud-Wide VM-Level Network Isolation**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

________________________________ Chair
Dr. A. Youssef

________________________________ External Examiner
Dr. Y. Liu

________________________________ Examiner
Dr. C. Assi

________________________________ Supervisor
Dr. Mourad Debbabi

________________________________ Co-supervisor
Dr. Lingyu Wang

Approved by ________________________________
Dr. Mourad Debbabi, Chair
Department of Concordia Institute for Information Systems Engineering

____________ 2017 ________________________
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

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

Yu Shun Wang

The multi-tenancy of a 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 thesis, 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 the compliance of isolation results against tenant-specific high level security policies.

Acknowledgments

I would like to express my gratitude to my supervisors Dr. Mourad Debbabi and Dr. Lingyu Wang for their guidance, advice and constant encouragement. I would also like to thank the Audit Ready Cloud (ARC) team for their enormous efforts.

Last but not least, heartfelt thanks go to my wife and my daughter. They are my greatest support and biggest comfort.

This thesis is partially supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under CRD Grant N01566.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

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 sharing physical resources, such as computing and networking services, among multiple tenants in an optimal and cost-effective way. 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, and cloud providers often have an obligation to provide clear evidences for sufficient network isolation, either as part of the service level agreements, or to demonstrate compliance with security standards (e.g., ISO 27002/27017 [2, 3] and CCM 3.0.1 [4]).

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

- 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 [5, 6]. Performing a cloud-wide verification of network isolation at the VM-level for such a cloud could potentially involve billions of VM pairs. 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 Chapter 2).

- 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 repetitively 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 [7].

- 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 [8]), 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.

Figure 1.1 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.

- The network isolation may be compromised through either unintentional misconfigurations

[1]To make our discussions more concrete, the examples will mostly be based on OpenStack, and Chapter 7 discusses the applicability of our approach to other cloud platforms.

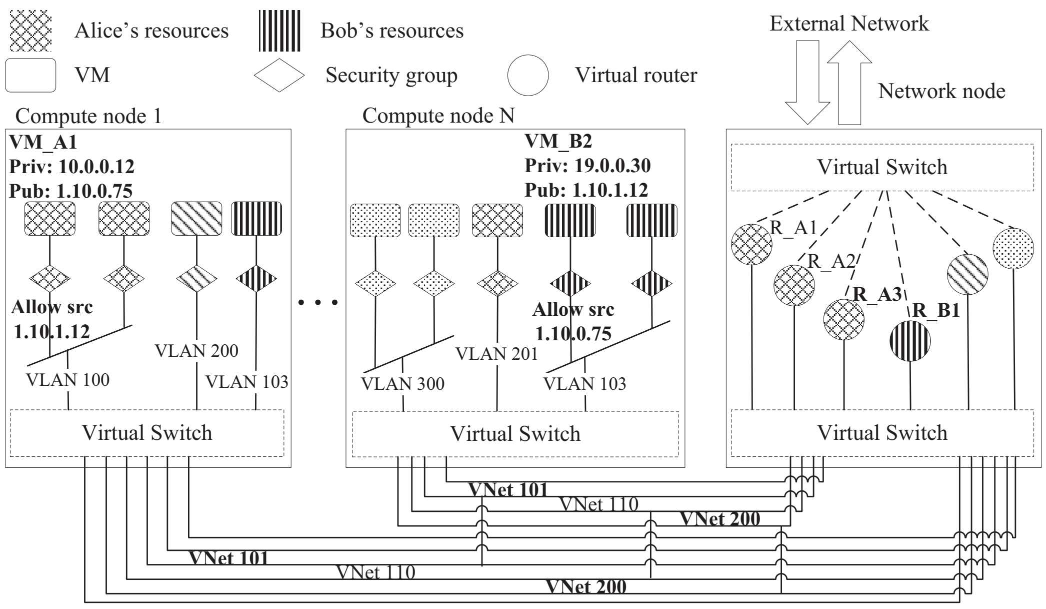


Figure 1.1: An Example of a Multi-Tenant Cloud

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 [9], 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 [10] 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/her 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 timely detect such a breach of network isolation, the challenge Alice faces is again threefold. First, assume the cloud has $25,000$ 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 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).

1.2 Contributions

In this thesis, we present *TenantGuard*, a scalable system for verifying cloud-wide, VM-level network isolation at runtime, while taking into consideration 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 trees [11] and X-fast binary tries [12]. Finally, TenantGuard leverages existing cloud policy services to check isolation results against tenant-specific high level security policies. The following summarizes our main contributions.

- To the best of our knowledge, this is the first system that can verify cloud-wide, VM-level 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 Chapter 6).
- 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 [8], 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.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews the related work. Chapter 3 describes our threat model and virtual network model. Chapter 4 discusses our system design. Chapter 5 provides the details of TenantGuards integration into OpenStack. Chapter 6 gives experimental results. Chapter 7 discusses the adaptability and integrity preserving. Chapter 8 provides limitations and future directions, and concludes the thesis.

Chapter 2

Related Work

Network verification works can be divided into two categories by their research objects, networks in non-cloud and cloud environments. In this chapter, we discuss them separately in section 2.1 and 2.2. Additionally, in section 2.3, we compare them with TenantGuard in terms of offered features and performance.

2.1 Non-Cloud Networks Verification

In the context of non-cloud networks, several works (e.g., [13, 14, 15, 16, 17]) propose network verification approaches on conventional networks, while others propose works on SDN (e.g., [18, 19, 20]). In their pioneering work, Xie et al. [13] propose an automated static reachability analysis of physical IP networks based on a graph-based model. Hassel[16] detects violations of network invariants such as absence of forwarding loops using geometric header space model, while Anteater [14] and ConfigChecker [15], offer similar features by representing forwarding rules as logical predicates which are able to be verified by model checkers. Although those works are successful for verifying enterprise and campus networks, they cannot address challenges of large scale cloud-based virtual networks, especially verifying network properties on VM-level granularity. For instance, Hassel [16] needs 151 seconds to compress forwarding tables before spending an additional 560 seconds in verifying loop-absence for a topology with 26 nodes. On the other hand, AP Verifier [17] improves the performance by extracting sets of predicates from relatively larger amount of forwarding rules, however, it works as 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 to tackle virtual networks in large scale clouds. Furthermore, most of these works consider routers/switches as the source and destination nodes for their verification, while we target VMs pairs which have much larger amount.

Other works (e.g., [18, 19, 20]) propose approaches for SDN networks, where larger amount of nodes are targeted. VeriFlow [18] and NetPlumber [19] (extension of Hassel [16]) outperform previous works by proposing a near real-time verification based on incremental method, where network events are monitored for configuration changes, and verification is performed only on the impacted part of the network. On the other hand, Libra [20] uses a divide and conquer technique to verify forwarding tables in large networks for subnet-level reachability failures, and enhance their performance leveraging map-reduce parallelization. Although incremental and parallel methods are applied to address larger scale of nodes, all of them are designed for centrally managed SDN, which is different from autonomously governed virtual networks among public clouds. Hence, some existing limitations might prevent these works from being directly applied to virtual networks verification in clouds. For example, as mentioned in their discussion [18], VeriFlow has difficulty to address NAT inside end-to-end path(which is prevalent in public clouds) due to they rely on equivalent classes model obtained from prefixes, while the scalability of NetPlumer [19] is limited by their Linear Fragment assumption on forwarding rules, which may be hold by centralized network management policies and deployments, but may not in multi-tenant clouds as each tenant may have different networking background, knowledge and management style. As for Libra [20], their divide-conquer scheme is based on their assumption that prefixes used in forwarding rules cannot be more specific than CIDR of subnet, which only can be guaranteed by specific deployments, since forwarding rules violating such assumption are legitimate in ordinary routers. Even so it is prone to violations caused by unintentionally made rules. Therefore, their technique will face severe challenges in the context of autonomous governed virtual networks in public clouds, where such requirement is hard to be enforced by all tenants.

2.2 Cloud-Based Network Verification

In the context of cloud-based network verification, there are several works (e.g., [21, 7]) aim to design general security property verification engines over physical networks in data centers owned by cloud providers, where virtual networks created by tenants (which is our main research object), are not take into account. However, NoD [21] and its successor [7] provide all-pair VM reachability feature over large-scale physical networks that can be compared with our work in terms of performance. We consider these works as complementary to our approach as they tackle the physical network verification in large cloud data centers from cloud providers view, while our effort is focused on tenant's virtual networks that are built on top of these physical networks and can be at the origin of several security issues due to their high dynamism and versatility.

On the other hands, several works (e.g., [22, 23, 24]) address security property verification over virtulization infrastructures. Bleikertz et al. [23] propose CloudRadar, a static information flow analysis system dedicated on VMware platform, where only a few layer-2 virtualized network elements such as vswitch and vlan are discussed, while higher level virtual network elements for packet forwarding and filtering such as virtual router and firewalls are absent. Bleikertz et al.[22] and Probst et al. [24] propose verification on filtering rules in security group and firewall's network access control respectively, but none of them can verify VM reachability due to the lack of capturing and analyzing packet forwarding rules through virtual routers inside multi-tenant clouds.

Congress [25] 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, which are difficult to solve and are not supported by Congress [25]. Therefore, we integrated TenantGuard into Congress in order to check network isolation results provided by TenantGuard against tenants' security policies defined in Congress. 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.

Network	Proposals	Methods	Features						Physical vs Virtual net.	
			Paral.	Incr.	NAT	All Pairs Reach.	Loop	Black Hole	Phys.	Virt.
Non-Cloud	Hassel [16]	Custom			●		●	●	●	
	NetPlumber [19]	Graph	●	●	●	●	●	●	●	
	Anteater [14]	SAT solver	●		●		●	●	●	
	Veriflow [18]	Graph		●			●	●	●	
	AP verifier [17]	Custom		●	●		●	●	●	
	Libra [20]	Graph	●	●	●		●	●	●	
Cloud	NoD [21]	SMT Solver			●	●	●	●	●	
	Nod with symmetry and surgery [7]	SMT Solver			●	●	●	●	●	
	CloudRadar [23]	Graph		●						●
	[24]	Graph			●					●
	TenantGuard	Graph	●	●	●	●	●	●		●

Table 2.1: Comparison of Features Offered by Existing Solutions

The symbol (●) indicates that the proposal offers the corresponding feature

2.3 Comparison with TenantGuard

Table 2.1 summarizes the qualitative 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. The next column compares those works to TenantGuard on various features, e.g., the support of parallel implementation, incremental verification, NAT, and all pairs reachability verification (which is the main target of TenantGuard), detection of forwarding loops and black holes. The last column compares the scope of those works, i.e., whether each work is designed for physical or virtual networks.

Network	Proposals	Size of input			Verif. Time
		VMs	Routers	Rules	
Non-Cloud	Hassel [16]	-	26	756.5k	-
	NetPlumber [19]	-	52	143k	60
	Anteater [14]	-	178	1,627	-
	Veriflow [18]	-	172	5millions	-
	AP verifier [17]	-	58	3,605	-
	Libra [20]	-	11,260	2,650k	-
Cloud	NoD [21]	100,000	-	820k	471,600
	NoD with symmetry and surgery [7]	100,000	-	820k	7,200
	CloudRadar [23]	30,000	-	-	-
	[24]	23	-	-	-
	TenantGuard	100k	1,200	850k	1,056

Table 2.2: Comparison of Existing Solutions Performance

All verification time measurements are reported in seconds.

On the other hand, in table 2.2, quantitative comparison is given in terms of input size and verification time among these works. With regard to input size, for works on cloud environment, number of VMs, routers (or switches), forwarding and filtering rules are listed, while for those non-cloud works, the latter two are indicated. As for the verification time, only results on all-pair reachability are reported. Note all these results come from tests under single-machine environment, although our work has implemented parallelization and shown significant speedup. We report our parallel test results in chapter 6.

From the table 2.1, we can see among the works dealing with virtual networks of clouds, our work is the only one addressing all-pair reachability, as well as loop and black hole detection with parallelization. Comparing to the works on physical networks with all-pair reachability feature, as shown in table 2.2, TenantGuard outperform others on similar input size.

Chapter 3

Models

In this chapter, we describe the threat model and propose our hierarchical graph model for cloud virtual networks.

3.1 Threat Model

We assume the cloud infrastructure management system may have implementation flaws or vulnerabilities that can potentially be exploited by a malicious cloud tenant or its users to violate the other tenants' network isolation polices. We assume the cloud provider and its infrastructure management system will ensure the integrity of its configuration data stored in databases and logs collected through API calls, event notifications, or database queries. Specifically, to ensure the integrity of information and event notifications collected from OpenStack, we can rely on existing techniques such as hardware-based platforms that ensure data secrecy and code integrity based on TPM (e.g, [26, 27]), and software-based tampering detection solutions (e.g., [28]). Also, we assume the vulnerabilities existing in OpenStack do not affect the integrity of collected data, logs and events notifications. Further discussion on existing methods to preserve integrity of the database and logs will be given in chapter 7. We assume each cloud tenant would provide security policies on network isolation to indicate which virtual resources should be accessible and by whom. Although our solution can catch a breach of such security policies (e.g., Bob can still access Alice's VM after Alice has tried to prevent it, as demonstrated in Chapter 1), the focus is not on detecting specific attacks or vulnerabilites (correlating our results with other security mechanisms, such as IDSs or vulnerability scanners, is an interesting future direction). We focus on the virtual network layer (layer 3) in this thesis, and our work is complementary to existing solutions at other layers (e.g., verification in physical networks or isolation with regard to covert channels caused by co-residency; more details are given in chapter 2). 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.

3.2 Virtual Network Model

To facilitate further discussions, we first define a hierarchical graph model to capture various components of virtual networks and their logical relationships. To build intuitions, we first provide an example.

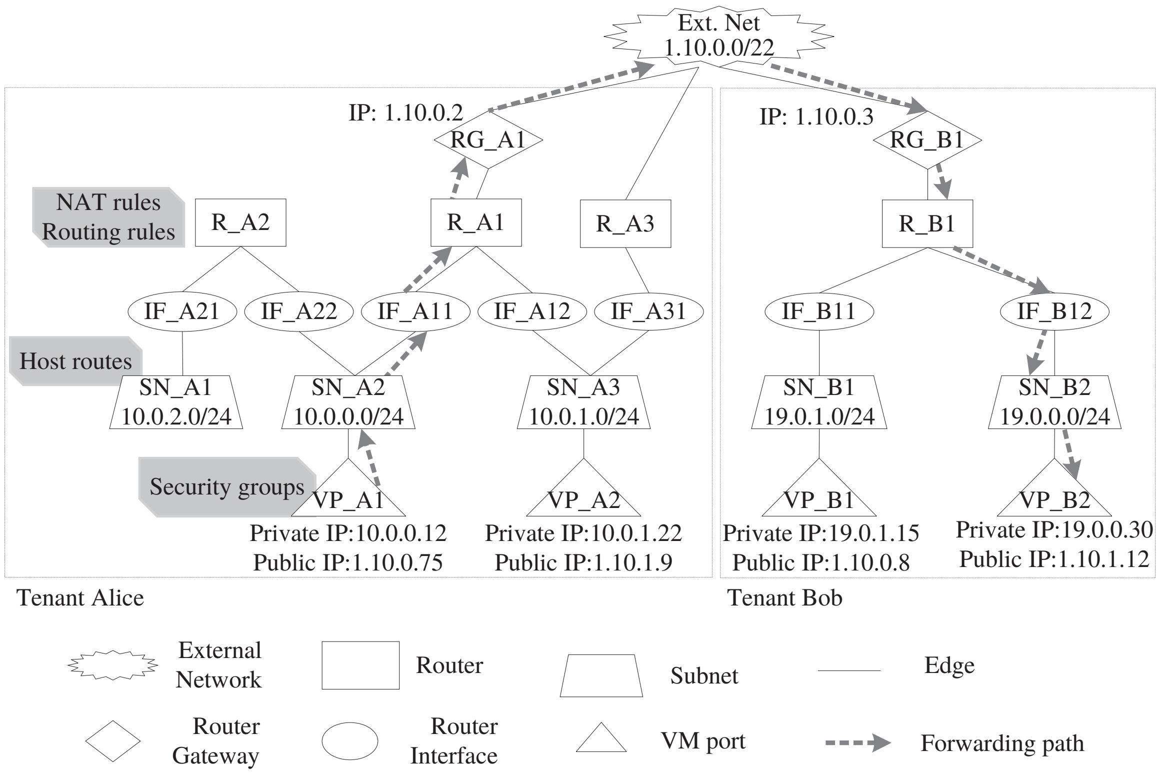


Figure 3.1: An Example of the Virtual Network Model

Example 1 *Figure 3.1 illustrates an instance of our model that captures the virtual networks of tenants Alice and Bob, following our example shown in Figure 1.1.*

- *Each tenant can create virtual subnets(e.g.,* `SN_A1` *and* `SN_A2` *of Alice). A subnet (e.g.,* `SN_A2`*) is collection of IP addresses represented as a CIDR (e.g.,* `10.0.0.0/24`*). Usually a default gateway is assigned (e.g., router interface* `IF_A11`*) to indicate by default where the packet sent from ports within this subnet should be forwarded to. In addition, tenant may specify a set of forwarding rules (host routes) to a subnet if it is connected to multiple routers and provide other options besides default gateway.*
- *A newly created VM (e.g.,* `VM_A1` *in figure 1.1) will be attached to a virtual port (e.g.,* `VP_A1` *in figure 3.1) 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 attached to VMs and act as virtual firewalls.*
- *Virtual routers (e.g.,* `R_A1`*) interconnect different subnets by router interfaces to route intra-tenant traffic(e.g., between* `SN_A2` *and* `SN_A3`*) and inter-tenant traffic, and direct them to external networks (e.g.,* `ExtNet_1`*) via router gateways (e.g.,* `RG_A1`*).*
- *Several interconnected external networks (only one in the figure 3.1) 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, public IP addresses allocated to VM (e.g.,* `VP_A1.Public_IP=1.10.0.75`*) should be used during the traversal.*
- *The mapping of private and public IP addresses are maintained by NAT rules located at virtual routers with a router gateway, where inbound traffic are filtered in such a way that packets are allowed only if their destination IPs match existing public addresses in NAT rules.*

More generally, Figure 3.1 may be abstracted as an undirected graph with typed nodes, as defined in the following.

Vertex Type	vm_port	subnet	v_router	v_router_if	v_router_gw	ext_net
vm_port	-	*	-	-	-	-
subnet	*	-	-	*	-	-
v_router	-	-	-	*	*	-
v_router_if	-	*	*	-	-	-
v_router_gw	-	-	*	-	-	*
ext_net	-	-	-	-	*	*

Table 3.1: Edges Between Vertex Type Pairs In Virtual Network Model

'*' represents existence of edges between vertex type pairs, while '-' means an absence of edges

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* $s = \{id, tenant\_id, Public\_IP, Private\_IP, type, rules\}$*, while* $type \in \{vm\_port, subnet, v\_router, v\_router\_if, v\_router\_gw, ext\_net\}$*, representing VM port, subnet, router, router interface, router gateway, and external network, respectively. E is a set of undirected edge representing the logical connectivity among those network components.*

Table 3.1 shows the matrix of edge existence among vertex type pairs, meanwhile table 3.2 demonstrates relationships between vertices reflected by these edges. The underlying logic follows how these virtual network components are created, as discussed in example 1. Consequently, this graph model is capable of reflecting network flows in virtual networks. For instance, as shown in table 3.1 and 3.2, since vertex representing a subnet is incident on edges joining all adjacent VM ports associated with the same subnet ID, for any pair of these ports there exists one path that passes the subnet vertex, which implies that traffic between these ports can be delivered at layer 2 level (subnets) without any help from devices working on layer 3 such as routers. On the contrary, traffic between ports located in different subnets will go through virtual routers if these subnet vertices are adjacent to the corresponding router interfaces. Note that table 3.1 shows edges exists between external network vertices. Further, these edges exist for any pair of distinct external network vertex, as shown in table 3.2. The reason behind this is that external networks consist of public accessible IP addresses, so any pair of them are considered to be reachable, which can be simply represented via these edges. As mentioned in section 2.2, works such as NoD [21] addressed

Edge	**Relationship**	**Description**
subnet-vm_port	One-to-Many	VM ports have identical subnet ID
subnet-v_router_if	One-to-Many	Router interfaces have identical subnet ID
v_router-v_router_if	One-to-Many	Router interfaces have identical router ID
v_router-v_router_gw	One-to-One	Router gateway belongs to the router
ext_net-v_router_gw	One-to-Many	Router gateways have identical external network ID
ext_net-ext_net	Many-to-Many	Any pair of distinct external network vertex has an edge

Table 3.2: Vertex Relationship Represented by Edges

Vertex Type	**VM Port**	**Router Interface**	**Router Gateway**
Degree	1	2	2

Table 3.3: Vertex Degree In Virtual Network Graph

reachability between physical networks in data centers of cloud provider, which can be abstracted as aforementioned external networks and their edges in our study. This is why we consider our work to be the supplement of these previous works. As a result, the subgraph connecting all external network vertices is a *complete graph* in our model. Additionally, G is a *simple graph* where there is no loop in which one edge joins a vertex to itself.

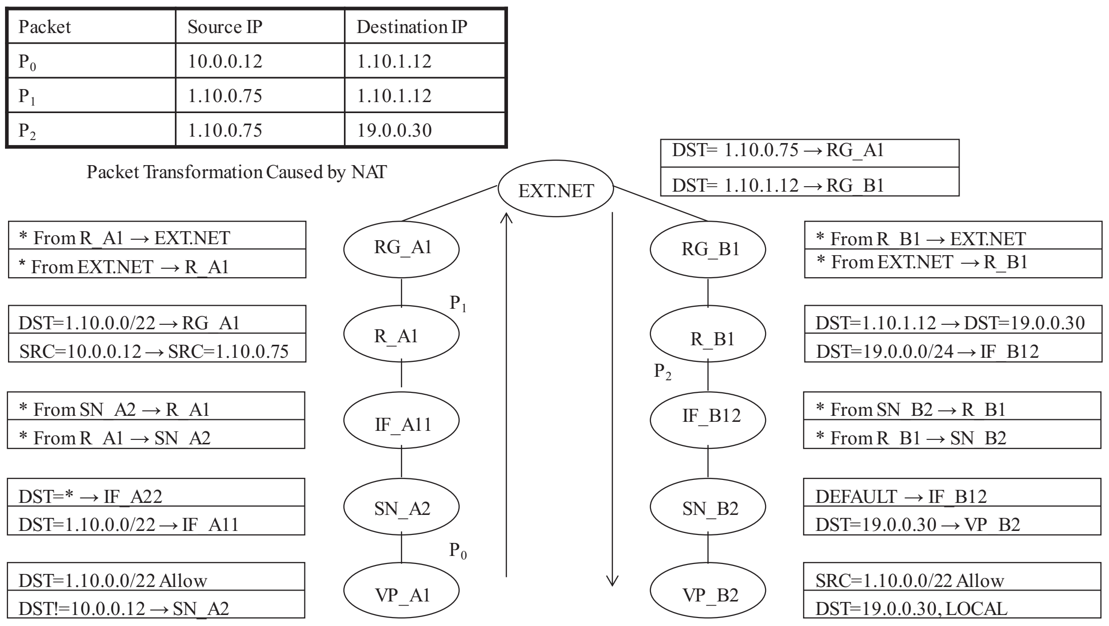
From the table 3.1 and 3.2, we can see that some types of vertices have fixed degrees in our graph model, which are summarized in table 3.3. Each VM port only has one edge joining the corresponding subnet, hence, its degree is 1 which implies these vertices are endpoints of the graph. Further, Router interfaces and gateways are only incident on two edges, one joining the associated routers, another linking subnet and external network respectively. Therefore, both of them have degree 2.

Based on table 3.3, G can be decomposed into a set of maximally connected sub-graphs [29] (denoted by $C_i = (V_i, E_i)$ in later discussions) by removing all edges between router gateways and external networks. Since router gateways always have degree of 2, with above removal, their degrees will be reduced to 1, causing those subgraphs representing different tenants' private virtual networks, including VM ports, subnets, routers, its interfaces and gateways, are disconnected from external networks. Imagine in figure 3.1, by such removal, Alice and Bob's private virtual networks is disconnected from each other, as well as from external networks.

With this graph model, we can have a topological view on virtual network components. To verify VM-level isolation, we need further analysis of forwarding and filtering rules attached to vertices in G, which is discussed in section 3.3.

3.3 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.



Packet	Source IP	Destination IP
P_0	10.0.0.12	1.10.1.12
P_1	1.10.0.75	1.10.1.12
P_2	1.10.0.75	19.0.0.30

Figure 3.2: An Example of Forwarding, Filtering Rules and NAT

Forwarding and Filtering

Network packets traversing virtual networks are typically governed by both filtering (security group rules) and forwarding (routing) rules, as illustrated in example 2.

Example 2 *Figure 3.2 shows a subgraph of the virtual network graph with associated rules which are extracted from figure 3.1 along the dotted line, as well as network address translation (NAT) occurred during the traversal from Alice's* `VM_A1` *to Bob's* `VM_B2`*, a typical inter-tenant traffic, where packet headers (only including source and destination IP) are listed in the upper table of* *figure 3.2. In this toy virtual network setup, the initial packet P_0, which is described in figure 3.2, is sent from* `VM_A1` *to* `VM_B2`.

- *P_0 is allowed by the egress security group attached to* `VM_A1` *, then forwarded to the subnet node* `SN_A2`.
- *For P_0, the most specific forwarding rules in* `SN_A2` *will make the packet sent to* `IF_A11` *according to longest matching semantics. Through* `IF_A11`*, P_0 will be delivered to router* `R_A1`.
- *At* `R_A1`*, P_0 hits the NAT rule on its source IP. As the result, sNAT in the virtual router will transform P_0 to P_1 by replacing the source IP of P_0 from* `VM_A1's` *private IP to its public IP 1.10.0.75. Sequentially, P_1 is forwarded to* `RG_A1` *according to the forwarding rules in* `R_A1`*. Afterwards, it is sent to the external network* `EXT.NET`.
- *P_1 traverses external networks to router gateway* `RG_B1`*, and arrives at router* `R_B1`.
- *Inside* `R_B1`*, $P1$ hits the NAT rule on its destination IP. Therefore, dNAT rewrites it with private IP of* `VM_B2` *19.0.0.30, denoted as P_2, and the packet is allowed to pass through the router to make it able to traverse Bob's private virtual network.*
- *Similarly, P_2 is forwarded to* `VP_B2` *and allowed by associated ingress security group rule. Now* `VM_B2` *receives the packet sent by* `VM_A1`*. Note with source IP in P_2,* `VM_B2` *can send responding packets to its peer reversely.*

Example 2 shows how both forwarding and filtering rules (security groups) with NAT dominate packets traversals inside virtual network graph. More generally, the following models the way that packets traverse virtual networks using a forwarding and filtering function, which captures these rules and NAT.

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

- *let $p \in \mathcal{P}$ be a symbolic packet 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 predecessor 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 $\{(p'_i,(v,w_i))\}_{i\in N}$, 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 $\{v,w_i\}_{\forall i\in N} \in E$.*
- *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 graph model, forwarding and filtering function just defined, we can formally model network isolation and related properties as follows.

Definition 3 *Given a virtual network graph $G=(V,E)$,*

- *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 Chapter 4 can also identify forwarding loops and blackholes as anomalies in virtual networks.

Chapter 4

TenantGuard Design and Implementation

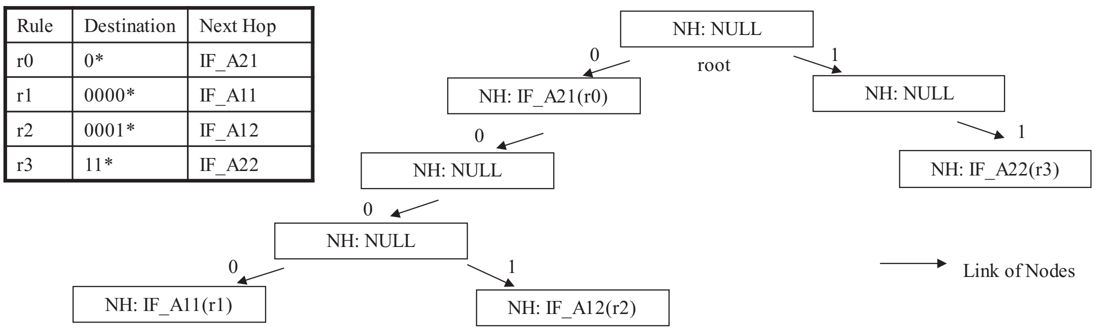
In this chapter, based on models illustrated in chapter 3, we introduce our methodology on VM-level network isolation verification. First, we propose an VM-level baseline solution directly applying the virtual network graph model and fd_G derived from definition 2. Second, we analyze prefix-level reachability and address challenges. Finally, we demonstrate TenantGuard, an three-step verification solution based on prefix-to-prefix reachability, in details. Note that incremental verification is not discussed in this thesis.

4.1 Baseline Solution

As demonstrated in section 3.3, forwarding a packet from one VM to another can be abstracted to a traversal through virtual network graph G. Given a VM port vertex u in G, a question that which VMs can reach u could be generalized to find the permutation of forwarding paths from other endpoints representing VMs in G to u. Like other graph traversals, forwarding paths are edge sequences in G. But the different point is how to choose an edge at vertices during traversals. For example, algorithms for shortest path problem leverage weight of edges to make such decision. As for virtual network graph traversal, edge selection depends on forwarding and filtering rules associated with vertices. If a matching rule is found, an edge is selected and the traversal continues.

Otherwise, packet may be dropped. Hence the challenges of such traversals not only exist in the scale of graph, but also amounts of rules in vertices.

Radix tree [11] is one of data structures that can be used to encode forwarding rules and query routing results efficiently. IP prefixes which represent matching condition of rules are encoded into binary bits as nodes in the tree, while routing results are stored in labeled variables of nodes. Given a key such as a destination IP, querying the corresponding routing result can be achieved by binary comparison in radix trees following longest matching principle.



Rule	Destination	Next Hop
r0	0*	IF_A21
r1	0000*	IF_A11
r2	0001*	IF_A12
r3	11*	IF_A22

Figure 4.1: A Radix Tree with Encoded Forwarding Rules

Figure 4.1 shows a toy example of a radix tree with encoded forwarding rules in a virtual router. Typically, these rules consist of destinations (IP prefix) and next hops, as shown in the table of figure 4.1. An empty radix tree is initialized with a root node. When one rule say r0 in the figure is encoded into the tree, to represent its first bit `0`, a new node is inserted, as well as a left link at the root to connect the node. Similarly, to represent a bit `1` in a prefix, such as the first bit of rule r3, another node is inserted with a right link at its predecessor. As a result, for any rule, its destination prefix is encoded into a chain of nodes, while the corresponding next hop is labeled as a variable named NH at the end of this chain. In such a way, we can construct the radix tree encoding all four rules as shown in the figure 4.1. Each destination and next hop can be considered as a key-value pair. Within radix trees, keys are not directly stored, but represented by their locations.

To forward a packet in such a virtual router, the most specific forwarding rule should be found. With radix trees, this can be done by a binary search starting from the root node, where bits in the destination IP are used to be compared with the bits represented by nodes in the tree. For example, given a destination IP that its four most significant bits of are 0010 as the key to query the radix tree shown in figure 4.1, along the left branch from the root node, the process of binary comparison will terminate at the third node, as there is not a right link for the third bit of 0010. Clearly, only one matching rule r0 is found during the comparison. Hence, a packet with such destination IP should be forwarded to `IF_A21`.

Radix trees can also be applied to filtering rules in security groups. These rules are key-value pairs of destination and filtering results (allowed or denied). We can encode these rules into radix trees in a similar way, and store the filtering results in different variables. To query the value for a given key, a binary search is executed. Note that for filtering rules, the first-come first-serve principle is applied. Hence, the binary search process will stop when the first matching result is found.

Algorithm 1 Baseline Verification

```
1: Input: $G = (V, E)$
2: Output: VM-level Isolation Results
3: for $each\ u, v \in V \wedge u \neq v$ do
4:     $P_{(u,v)} = IP_u \times IP_v$
5:     $R = null$
6:     for $each\ p \in P_{(u,v)}$ do
7:         $s = (p, (null, u))$
8:         $S.append(s)$
9:         while $s \neq (\neg null, (v, null))$ do
10:            $s = fd_G(s)$
11:            if $s \in S$ then
12:                $R.append(loop)$
13:                Goto 6
14:            end if
15:            if ( then$s.p = null$)
16:                $R.append(blackhole)$
17:                Goto 6
18:            end if
19:            $S.append(s)$
20:        end while
21:        $(R.append(forwardingpath))$
22:    end for
23:    if $\exists forwardingpath \in R$ then
24:        result = reachable
25:    else
26:        result = isolated
27:    end if
28: end for
```

The advantage of this encoding scheme is that we can get constant worst execution time for one query, regardless how many rules are encoded inside a radix tree. As shown in our toy case, assuming prefixes at most have l bit, the worst searching time for given keys is $O(l)$. Hence, the scalability in terms of number of rules is improved.

By applying this data structure, we can construct forwarding and filtering function. Given a virtual network graph G, for each vertex representing a virtual router, a radix tree can be initialized and associated with that vertex, where the forwarding rules of that router are encoded. In the same way, radix trees can be established for subnets to encode host routes. As for security groups, two radix trees can be established with egress and ingress filtering rules respectively. On the other hand, to address NAT, a lookup table can be applied to build maps of private-public IP pair associated with router gateways. Therefore, given a forwarding state $(p, (u, v))$ as defined in definition 2, we can use destination IP in p as the key to query the radix tree of v. If the matching result is found in the radix tree and next hop is pointed to w, the next forwarding state is obtained as $(p', (v, w))$, where p' may be different from p if NAT is applied in vertex v. On the other hand, if there is no matching key in the radix tree, a special forwarding state $(null, (v, null))$ is generated, indicating that a blackhole is found at the forwarding path. This process can be repeatedly carried out until a forwarding path is identified.

With virtual networking graph G and fd_G based on radix trees, we propose our baseline solution to verify reachability properties in definition 3 for VM pairs. Pseudo codes in algorithm 1 demonstrate a high level structure of our baseline solution. (u, v) represents any VM vertex pair in G, where u is the source, and v is the destination. The sets of all IP addresses assigned to u and v are denoted as IP_u and IP_v respectively, including their private and public IP. We use the set $P_{(u,v)} = IP_u \times IP_v$ to represents a collection of packets that their source and destination IP addresses are taken from IP_u and IP_v respectively. For each $p \in P_{(u,v)}$, we construct initial forwarding state $s_0 = (p, (null, u))$ as the input of $fd_G()$ to generate the next forwarding state s_1. For any such a forwarding state s, we use $s.p$ to denote the packet extracted from that state, as well as $s.pre$ and $s.cur$ represent sender and current node. On the other hand, we use a list denote as S to keep trace of forwarding states. By calling $fd_G()$ recursively, either there is a forwarding state reaches $(p, (v, null))$ representing v can receive such a packet, or a black hole or a loop described in definition 3 is detected. Verification results are appended to a list R. For each VM pair, if there exists at least one forwarding path as definition 2, they are reachable; otherwise, they are isolated.

Although this baseline solution is able to verify VM-level reachability properties in clouds, there may exist redundant computations that reduce the efficiency of verification. For example, consider verifying reachability from one arbitrary VM to multiple VMs within the same subnet. Although these destination IPs are different, since they are in the same IP address block, it is likely to get same route in routers. As a result, these forwarding paths may be identical until packets arrive at the subnet of destinations. Whereas in the baseline solution, since these forwarding paths are verified individually, traversals from the source VM to the destination subnet are repeatedly carried out. Furthermore, if tenants cluster their VMs with high density in subnets, such repeated computation will get worse. In the next section, we will discuss how to exploit such similarity in forwarding paths and address associated challenges.

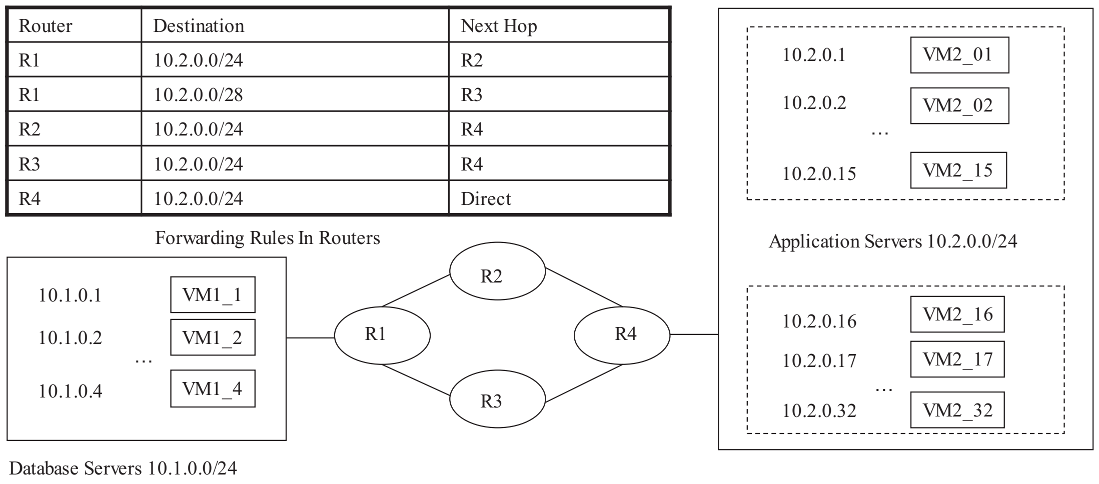
4.2 Prefix-to-Prefix Reachability

As mentioned before, forwarding paths from a VM to VMs within the same subnet may be similar, while the opposite may also be true. In this section, we use a concrete example to demonstrate both situations, then introduce our scheme to improve the verification efficiency, and keep the flexibility to handle challenges.

4.2.1 Challenges of Prefix-to-Prefix Reachability

One idea to avoid aforementioned redundant computation is to calculate forwarding path from subnet A to subnet B using the destination prefix as the key to query router's forwarding rules, then apply the prefix-to-prefix routing result to any VM pair in A and B before do further verification over filtering rules associated with security groups. But the problem is, this prefix-to-prefix forwarding path does not always have an unique result. We use an example shown in figure 4.2 to demonstrate the situation.

Example 3 *In figure 4.2, in one subnet with CIDR 10.1.0.0/24, four VMs running as database servers have IP addresses from 10.1.0.1 to 10.1.0.4. Meanwhile, within another subnet with CIDR 10.2.0.0/24, 32 VMs run as application servers that IP addresses range from 10.2.0.1 to 10.2.0.32. The two subnets are connected by 4 routers: R1, R2, R3, R4. The forwarding paths from 10.1.0.0/24*



Router	Destination	Next Hop
R1	10.2.0.0/24	R2
R1	10.2.0.0/28	R3
R2	10.2.0.0/24	R4
R3	10.2.0.0/24	R4
R4	10.2.0.0/24	Direct

Figure 4.2: An Example of Prefix-to-Prefix Reachability

to 10.2.0.0/24 are the target of our discussion, and the relevant forwarding rules are listed in the upper table.

If we omit the rule at the second row of the table temporally, then the prefix-to-prefix traffic has the unique forwarding path, which passes R1, R2 and R4 sequentially. Whereas, the tenant may consider let some parts of such traffic going through R3 for bandwidth or security concerns. When the second rule is applied, by the longest matching principle, traffic to application servers with IP ranging from 10.2.0.1 to 10.2.0.15 will pass R1, R3 and R4, while traffic to other application servers still go through R1, R2, R4. Therefore, the prefix-to-prefix forwarding path of the example splits into 2 branches at R1. Moreover, in virtual networks with larger scale, it is possible that these branches are split further if corresponding forwarding rules are configured at the successor routers. When such splitting propagates inside a large virtual network graph, it makes our verification work more complicated and challenging.

On the other hand, another challenge is about how to query radix trees using a prefix as the key instead of a single IP address. Furthermore, forwarding path splitting make it even worse as some destination IP addresses cannot be encoded into a prefix. For example, after such prefix-to-prefix forwarding path split at R1, the branch going through R2 have destination IP addresses ranged *from 10.2.0.16 to 10.2.0.255, which is a non-prefix segment that cannot be encoded into one CIDR. To query routing results in R2 for destination IP addresses ranged from 10.2.0.16 to 10.2.0.255, an iteration over the range is not so helpful, since there is no significant difference in terms of efficiency with our baseline solution.*

In summary, as shown in example 3, one main challenge of prefix-to-prefix forwarding path verification is how to manage propagation of path splitting in virtual networking graph, while another one is how to query radix trees efficiently with different forms of keys, including prefixes and non-prefix segments.

4.2.2 Ranges and X-Fast Trie

A fundamental step to address challenges of prefix-to-prefix forwarding path is how to represent IP prefixes and non-prefix segments. In our context, both of them can be considered as collections of continuous integers, since each IP address can be converted into an integer with fix length bits. For instance, an IP v4 address can be represented as a 32 bits integer in its binary form. Therefore, IP prefixes and non-prefix segments can be represented as ranges that each of them can be described by two integers, its lower and upper bounds. For example, the prefix 10.2.0.0/24 can be represented by integers corresponding to IP addresses 10.2.0.0 and 10.2.0.255, while the non-prefix segment described in example 3 can be represented by integers corresponding to 10.2.0.16 and 10.2.0.255 respectively. One range may only contain one integer when the two bounds are identical, which can reflect to special prefixes such as 10.2.0.1/32 which represents a single IP.

As shown in example 3, the destination prefix 10.2.0.0/24 can be considered as the input of fd_G at R1. After forwarding path splitting caused by rules in R1, 10.2.0.0$\sim$10.2.0.15 and 10.2.0.16$\sim$10.2.0.255 become the inputs of fd_G at R3 and R2 respectively. In another word, the initial destination range is divided into subranges as forwarding paths split. If such splitting propagates further in the virtual network graph, then those subranges will be divided into more specific subranges. On the other hand, to verify VM-level reachability, for any given VM IP within the prefix, we need to locate the proper subrange including that IP to obtain the routing result. Therefore, to address forwarding paths splitting, it is desirable that the data encoding scheme applied to ranges has following features:

- Subranges can be inserted to its parent range, as well as its associated routing results.
- Given any integer inside a range, there exist an efficient searching algorithm to find routing results associated to the most specific subrange where that integer falls into.

X-fast binary trie [12] is a good candidate to fulfill above requirements. It is a data structure to encode integers with fixed length l bits into a l-level binary trie (Note that integers in radix trees have variable length of bits). Hence, for any range representing the destination prefix, we can initialize a x-fast trie and insert two integers corresponding to the bounds of that range, while its routing result could be written into the both leaves. When the range is divided into subranges, the boundary integers of subranges are inserted to the same trie as well as the routing results to leaves. If forwarding path splitting propagates in the virtual network graph, more boundary integers will be inserted into the same trie. For each subrange, at most two integers are encoded in the trie, regardless how many elements that subrange has. As for the extreme case of ranges with identical bounds, only one leaf is inserted, but labeled as upper and lower bound either. As a result, for a given destination IP prefix, all forwarding paths can be stored and managed in one x-fast trie.

On the other hand, in x-fast tries, given any integer within the range, searching algorithm will fall into the leaf that corresponds to the boundary integer of the most specific range to the given key. This will ensure the proper routing result will be found for any VM with a specific destination IP. Further more, such searching algorithm has constant worst execution time regardless how many subranges the trie has due to x-fast trie's root-to-leaf binary searching path. With hash table for each level in x-fast tries, the worst searching time is $O(log(l))$, better than the ordinary one $O(l)$. Therefore, x-fast tries can offer scalability and efficiency for VM-level forwarding path verification.

The example shown in figure 4.3 demonstrates how IP prefixes are encoded into a X-Fast Trie with corresponding routing results.

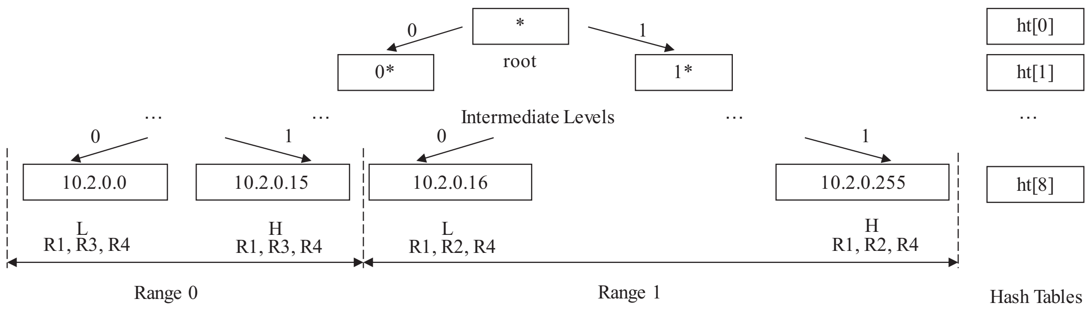


Figure 4.3: Ranges Encoded into a X-Fast Trie

Example 4 *We use the same virtual network setting in example 3. As analyzed, there is a forwarding path splitting at R1 from database server subnets(10.1.0.0/24) to application server subnets(10.2.0.0/24). A x-fast trie is initialized with the destination prefix 10.2.0.0/24 to encode subranges and its routing result. At first, two leaves are inserted, which represents integers corresponding to 10.2.0.0 and 10.2.0.255, labeled as 'L'(lower bound) and 'H'(upper bound) respectively. Note the level of the trie in figure 4.3 is not 32, which is the length of IP V4 addresses. It is caused by implementation details and will be explained in section 5.1.*

Since a forward path split occurs at R1 due to the forwarding rule, a new range starting with 10.2.0.0 and ending with 10.2.0.15 should be written into the trie. Then a range insertion method is performed with corresponding integers. Note that new range is overlapped with the initial one, hence another leaf corresponding to 10.2.0.16 is also inserted, while forwarding results are synchronized between lower and upper bounds. Thus, as the final results, range 1 representing 10.2.0.0/28 have two leaves with forwarding result R1, R3, R4, while range 2 have two leaves containing result R1, R2, R4, which represents IP address set $a \setminus b$, where a and b are set of IP address 10.2.0.0/24 and 10.2.0.0/28 respectively. Note that hash tables for each level of the trie is updated immediately with leaf insertion. When a routing result search for an individual IP is performed, values stored in those hash tables are used to compare with the corresponding part of the key to reduce searching time. With this trie, given a IP say 10.2.0.31 as the key, the routing result will be fetched from the leaf corresponding to range 1, since this range is the most specific one for 10.2.0.31.

From the example 4, we can see that by encoding routing results of range and its subranges into a x-fast trie, forwarding path splitting in virtual network graph, the first challenge of prefix-to-prefix reachability verification, could be well managed.

4.2.3 Query Radix Trees Using Ranges As Keys

As discussed in section 4.2.1, another challenge in the prefix-to-prefix reachability verification is how to query radix trees associated with virtual routers to get routing results using IP prefixes and non-prefix segments as keys. Now we introduce our searching method in radix tree addressing this challenge.

Conventionally, querying routing result in radix trees uses a single key, such as a destination IP address in our baseline solution. In prefix-to-prefix reachability verification, ranges including prefixes and non-prefix segments should be used as keys. The naive way that iterates all the elements in ranges should be excluded, as it will make no gain in terms of efficiency compared with our baseline solution.

At first we examine the case that keys are prefixes. Comparing with single IP as a key, the search for a prefix is quite similar on bit comparison through radix tree. The difference is that when keys are prefixes, all matching values should be gathered in such a way that the relationship between routing results and its corresponding subranges of destination should be clearly reflected. For this purpose, a method mixed with binary comparison and preorder traversal is developed, which is demonstrated in the example shown in figure 4.4.

Example 5 *In figure 4.4, the same radix tree in figure 4.1 is used, while 0* is the key to query all matched forwarding rules encoded in this tree. The dashed arrows labeled with numbers show the access sequences. At first, starting from root node, binary comparison is executed for all bits in the key, and the most specific one is recorded. Obviously, in this tiny example only one bit 0 is compared shown as step 1, and r0 is the matched result. Then, starting from the node representing 0*, a preorder traversal is performed over the subtree in a recursive way such that the left branch is firstly traversed, then go back to the upper node and start traversal in the right branch. Such process is presented as step 2 to 9, where two matching rules r1 and r2 are found. Finally, the process is terminated when it is returned from right branch of the starting node. By this combined*

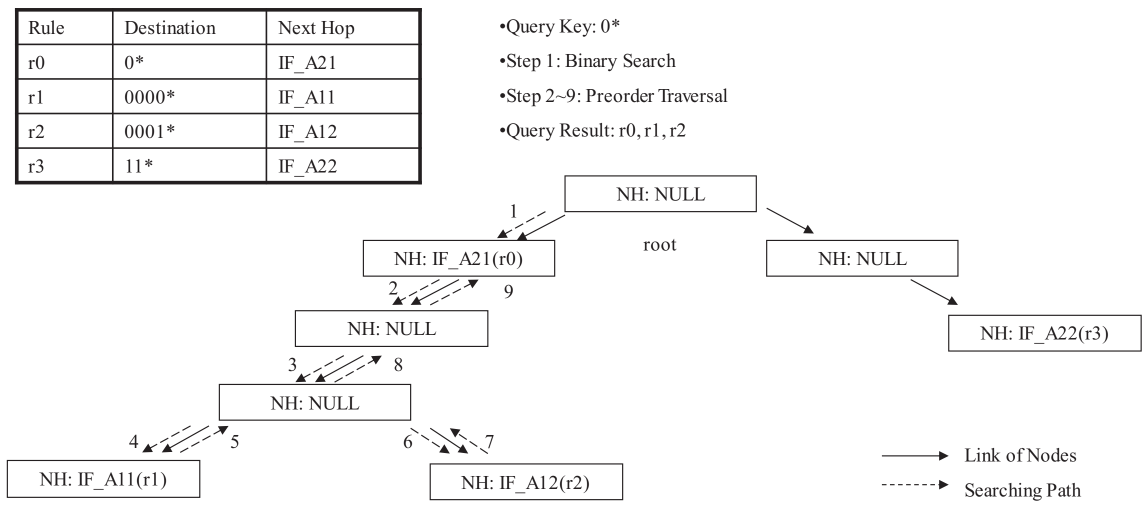


Figure 4.4: Querying Matched Rules in a Radix Tree Using a Prefix as the Key

Rule	**Prefix**	**Subrange**	**Next Hop**
r0	0*	[32:127]	IF_A21
r1	0000*	[0:15]	IF_A11
r2	0001*	[16:31]	IF_A12

Table 4.1: Subranges And Routing Results In Example 5

method, all matching rules(r0, r1, and r2 in the example)are gathered. Note the traversal will never reach r3, because the process never go beyond the upper part of the preorder starting node. Hence, these rules that does not match the key will never be gathered as the result. All routing results are recorded, as well as the corresponding root-to-node bit sequences that can be converted to subranges. To make subranges more readable, we use pseudo 8-bit IP addresses in this example. Therefore, prefix 0 represents IP address space [0:127], while 0000*, 0001* represents [0:15] and [16:31] respectively. Then the result of query with 0* can be shown as in table 4.1, with the next hops and its corresponding subranges. Note that the first result corresponds not to [0:127] but to subrange [32:127], due to the fact that r1 and r2 are more specific than r0.*

As shown above, query radix trees using prefixes as keys can be done by the method that combines binary comparison and preorder traversal. Notice in the first row of table 4.1, a non-prefix subrange [32:127] is generated and it will be the key to query the radix tree of the virtual router say

Rule	Destination	Next Hop
r0	*	IF_A31
r1	0101*	IF_A41

Table 4.2: An Excerpt of Forwarding Rules In Example 6

R_A2. Next, we discuss how to tackle such a non-prefix key in the following example.

Example 6 *Sample forwarding rules in the virtual router R_A2 are given in the table 4.2. Bit comparison cannot be directly applied since the key [32:127] can not be interpreted as a prefix. The trick is done by two steps. At first, we use the original prefix 0* as a key to query the routing results in the radix tree of R_A2, with the exact method in example 5. Then we get following result:*

- *for IP ranged in [0:79], the packet should be forwarded to IF_A31 by the rule r0*
- *for IP ranged in [80:95], the packet should be forwarded to IF_A41 by the rule r1*
- *for IP ranged in [96:127], the packet should be forwarded to IF_A31 by r0*

In the next step, we extract the results matching the non-prefix key. Thus, we have following routing results for the key [32:127]:

- *for IP ranged in [32:79], the packet should be forwarded to IF_A31*
- *for IP ranged in [80:95], the packet should be forwarded to IF_A41*
- *for IP ranged in [96:127], the packet should be forwarded to IF_A31*

We call this two-step method as *range copy*. With range copy, we successfully gather all matching result in radix trees for a non-prefix key without iterating all elements in that key. Moreover, given a virtual router and a destination prefix, only one query is needed for all subranges to do such range copy. Consequently, the result of that query can be cached for reusing.

With the two methods demonstrated in example 5 and 6 altogether, the second challenge of prefix-to-prefix reachability is addressed.

4.3 TenantGuard Design

Based on prefix-to-prefix reachability discussed in section 4.2, TenantGuard, a verification solution of cloud-wide VM-level network isolation, is presented in this section. At first, we give an overview on its three-step verification; then each step is introduced in detail; finally, we analyze the complexity.

4.3.1 Three-Step Verification Overview

TenantGuard leverages the hierarchical virtual network model presented in Chapter 3 to partition 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), such that the verification can be performed in three steps. As we will confirm with experimental results in Chapter 6, such a decomposition significantly improves TenantGuard's performance in a cloud-wide, VM-level verification of network isolation.

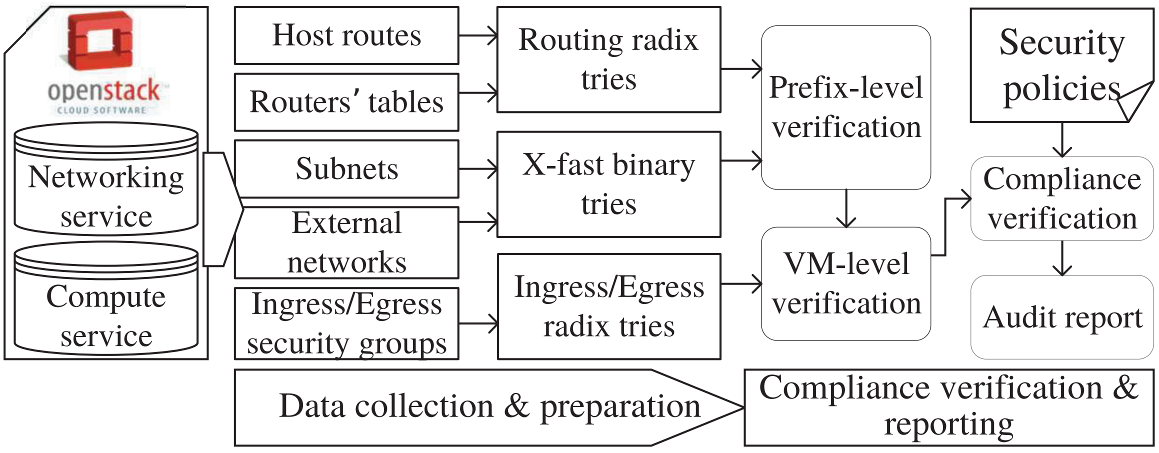


Figure 4.5: An Overview of TenantGuard

Figure 4.5 provides an overview of TenantGuard solution. Input data from the cloud infrastructure management system, such as virtual routers, subnets, VMs and its ports, are collected and loaded into a virtual network graph described in Chapter 3 , while router rules, subnets' host routes, and security groups rules, are encoded into radix trees associated with vertices in the graph. Then, prefix-level verification is carried out using X-fast binary tries. Afterwards, VM-level isolation is verified based on prefix-level routing results stored in X-fast tries and its associated filtering rules encoded in radix trees. Further, VM-level isolation results can be the input of compliance verification, compared with the tenant's pre-defined security policies(defined in Congress) to generate audit result.

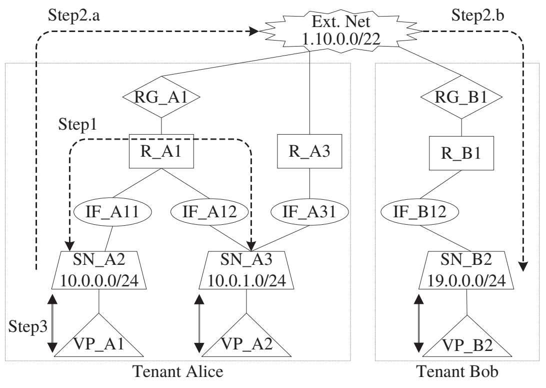


Figure 4.6: An Example of the Three-Step Verification

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

Example 7 *Figure 4.6 illustrates the three-step verification using our running example shown in Figure 3.1.*

- *In* `Step 1` *(detailed in Section 4.3.2), prefix-level isolation verification within the same components/sub-graph is performed. For instance, the isolation between* `SN_A2` *and* `SN_A3` *through the router* `R_A1` *belonging to tenant Alice is verified using their respective private IP prefixes (e.g.,* `10.0.0.0/24` *and* `10.0.1.0/24`*).*

- *In* `Step 2` *(detailed in Section 4.3.2), 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 first subnet as source(e.g.,* `SN_A2`*) and the external network, and* `Step 2.b` *for verifying isolation between the external network and the second subnet as destination (e.g.,* `SN_B2`*). This verification also involves public and private IP NAT.*

- *Finally,* `Step 3` *(detailed in Section 4.3.3) performs VM-level security groups verification for any pair of subnets found to be reachable using* `Step 1` *and* `Step 2`.

Algorithm 2 *TenantGuard*

```
1: Input: $G = (V, E)$
2: Output: VM-level isolation results
3: for each $C_i$ in $G$ do
4:     for each $s_{ij}$ in $C_i$ where $s_{ij}.type = subnet$ do
5:         for each $s_{ik} \neq s_{ij}$ in $C_i$ do
6:             $BTrie_{ik} = initBTrie(s_{ik}.CIDR, s_{ij}.id)$
7:             $prefix\text{-}to\text{-}prefix(BTrie_{ik})$
8:         end for
9:         for each $v \in V | v.type = ext\_net$ and $\exists$ a path from $s_{ij}$ to $v$ do
10:            $BTrie_{ij} = initBTrie(v.CIDR, s_{ij}.id)$
11:            $prefix\text{-}to\text{-}prefix(BTrie_{ij})$
12:        end for
13:        for each $r, rg \in V | r.type = router, rg.type = v\_router\_gw, \{r, rg\} \in E, \exists$ a path from $s_{ij}$ to $r$ do
14:            $BTrie_{ij} = initBTrie(s_{ij}.CIDR, r.id)$
15:            $prefix\text{-}to\text{-}prefix(BTrie_{ij})$
16:        end for
17:    end for
18: end for
19: for each pair $(VM_{src}, VM_{dst})$ do
20:     $VM\text{-}to\text{-}VM(VM_{src}, VM_{dst})$
21: end for
```

In addition, Algorithm 2 shows a high-level algorithmic view of TenantGuard. It takes the virtual network graph G as its input. The function *initBTrie* initializes X-fast binary tries with destination prefixes, while the function *prefix-to-prefix* (Detailed in Algorithm 3 in Section 4.3.2) employs X-fast binary tries to verify prefix-level isolation between all of source and destination IP prefixes pairs. Lines 5-8 implement `Step 1` in example 7, where C_i is a maximally connected sub-graph. Next, lines 9-16 realize `Step 2.a` and `Step 2.b` to verify prefix-level isolation between subnets and external networks. Finally, lines 19-21 carry out `Step 3`, where the function *VM-to-VM* checks VM-level isolation based on their security groups (see Algorithm 4 in Section 4.3.3).

4.3.2 Prefix-Level Verification

Now we detail the function *prefix-to-prefix* that performs prefix-level verification.

The function *prefix-to-prefix* takes an initialized x-fast trie as its input, where two leaves representing lower and upper bounds of the destination prefix are inserted by calling the function *initBTrie*, as shown Algorithm 2. A set of variables is initialized within each leaf, including B, RLB, HR, etc.

- Variable B stores the boundary of the IP ranges for each leaf. Its value is either L for the

Algorithm 3 *prefix-to-prefix*$(btrie)$

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

lower bound, H for the upper bound, or LH if a single leaf with a specific IP address (e.g., 1.10.0.2/32).

- 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. When an x-fast trie is initialized with the prefix, RLB is 00 within the both bounds.

- 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. Initial values of HR are obtained from the query result over the source subnet's radix tree with the key taken from the destination prefix, since the rules encoded in that radix tree decide which router should be chosen as the next hop of the source subnet.

As shown in Algorithm 3, the function *prefix-to-prefix* recursively locates leaf nodes with RLB labeled as 00 and the corresponding subrange of destinations, extract the current router from variable HR, and queries the corresponding radix tree using the destination prefix as the key. A temporary trie is used to encode the query result, then the routing results corresponding to the current subrange are copied to $btrie$. Note that in Algorithm 3 Line 6-10 show that these temporary tries can be cached and reused. The function is terminated at the absence of leaves with RLB equal to 00 in

$btrie$, which means the prefix-to-prefix reachability is identified for all subranges of the destination prefix. The core of this algorithm is the $Match$ and $Copy$ functions, which are the implementations for the methods regarding querying radix trees using ranges as keys, as discussed in Section 4.2.3.

We can use the function *prefix-to-prefix* to verify prefix-level isolation on each hop between all pairs of source and destination IP prefixes. It is clear that 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 $btrie$ to update the new results according to the routing results over the node's radix tree using each IP range as the query key. Hence, for any prefix pair, such $btrie$ helps to find routing results for every forwarding path branch in the virtual network graph, and each routing result is encoded into leaves representing the corresponding subrange, which is ready for querying by any individual IP in the range.

Now we use an example to show the progressive update inside the $btrie$ using *prefix-to-prefix* to verify prefix reachability with the virtual network setting in Figure 4.2.

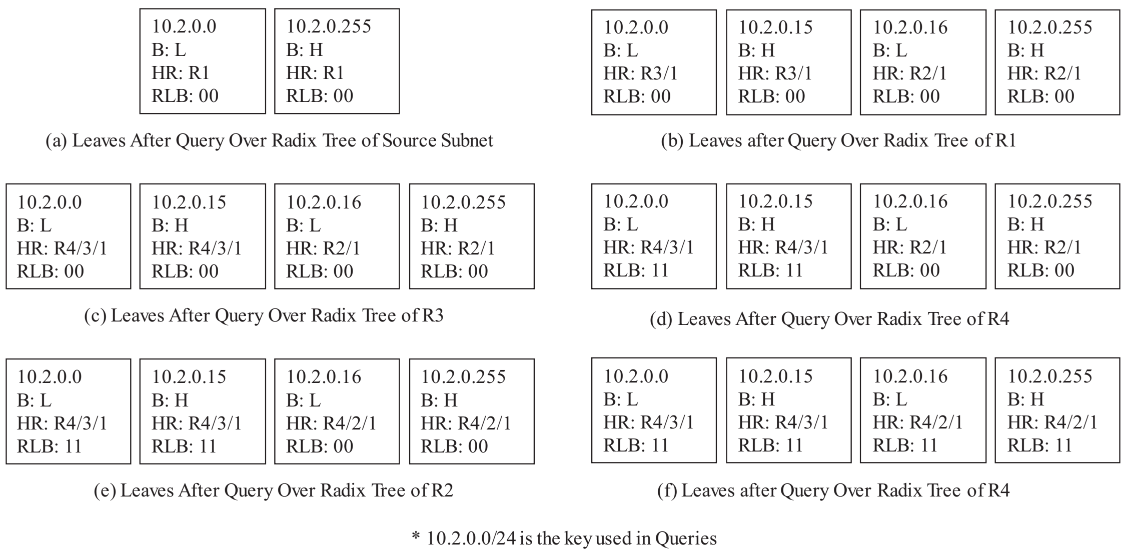

10.2.0.0 B: L HR: R1 RLB: 00
10.2.0.255 B: H HR: R1 RLB: 00
(a) Leaves After Query Over Radix Tree of Source Subnet
10.2.0.0 B: L HR: R3/1 RLB: 00
10.2.0.15 B: H HR: R3/1 RLB: 00
10.2.0.16 B: L HR: R2/1 RLB: 00
10.2.0.255 B: H HR: R2/1 RLB: 00
(b) Leaves after Query Over Radix Tree of R1
10.2.0.0 B: L HR: R4/3/1 RLB: 00
10.2.0.15 B: H HR: R4/3/1 RLB: 00
10.2.0.16 B: L HR: R2/1 RLB: 00
10.2.0.255 B: H HR: R2/1 RLB: 00
(c) Leaves After Query Over Radix Tree of R3
10.2.0.0 B: L HR: R4/3/1 RLB: 11
10.2.0.15 B: H HR: R4/3/1 RLB: 11
10.2.0.16 B: L HR: R2/1 RLB: 00
10.2.0.255 B: H HR: R2/1 RLB: 00
(d) Leaves After Query Over Radix Tree of R4
10.2.0.0 B: L HR: R4/3/1 RLB: 11
10.2.0.15 B: H HR: R4/3/1 RLB: 11
10.2.0.16 B: L HR: R4/2/1 RLB: 00
10.2.0.255 B: H HR: R4/2/1 RLB: 00
(e) Leaves After Query Over Radix Tree of R2
10.2.0.0 B: L HR: R4/3/1 RLB: 11
10.2.0.15 B: H HR: R4/3/1 RLB: 11
10.2.0.16 B: L HR: R4/2/1 RLB: 11
10.2.0.255 B: H HR: R4/2/1 RLB: 11
(f) Leaves after Query Over Radix Tree of R4
* 10.2.0.0/24 is the key used in Queries

Figure 4.7: Leaves of a X-fast Trie in the Prefix-Level Verification

Example 8 *In this example, reachability from prefix 10.1.0.0/24 to 10.2.0.0/24 is verified. Figure 4.7 shows insertion and update of leaves in the x-fast trie. Three aforementioned variables are presented: B, HR and RLB. To make it more readable, bit-string of leaves are replaced with its* *corresponding IP address, meanwhile only router names are shown for HR in a reverse order of access. For example, HR R4/3/1 show a router access sequence R1, R3 and R4. When RLB is 00, the next hop will be R4. Note that the key 10.2.0.0/24 is used throughout queries.*

- *In step a, two leaves representing the range 10.2.0.0/24 are inserted. Based on the query result over radix tree of source subnet where host routes are encoded, since default gateway is R1, HR at both leaves are R1, while RLB with 00 indicates the forwarding path verification for this range is on going.*

- *In step b, since the query result over R1 radix tree leads to forwarding path split, two subranges need to be encoded into* `btrie`*: from 10.2.0.0 to 10.2.0.15, and from 10.2.0.16 to 10.2.0.255. For the former, its next hop is R3 while for the latter it is R2. Hence, HRs of leaves are updated as R3/1 and R2/1 respectively.*

- *Step c shows updating the first subrange after the query over R3 radix tree. Consequently, HRs are updated to R4/3/1 for the subrange.*

- *In step d, a query is executed over R4 radix tree. By forwarding rules in R4 the destination subnet is reached, since 10.0.2.0/24 is directly connected to R4. As the result, RLB is updated to 11 showing that the final result is reachable for the subrange.*

- *Since RLB of the first subrange is 11, prefix-to-prefix jumps to the next subrange, as shown in steps e and f. As this is a non-prefix subrange, range copy is used to update the result. Similarly, the forwarding path ends up with R4 while the final value of corresponding RLB is 11.*

Example 8 demonstrates how *prefix-to-prefix* is applied to `step 1` of TenantGuard. For `Step 2` it is quite similar. As aforementioned, in `Step 2.a`, public prefix of the external network is used to initialize a x-fast trie, while the graph traversal using *prefix-to-prefix* starts from the source subnet. On the other hand, in `Step 2.b`, another x-fast trie is initialized with the private prefix of the destination subnet, and the corresponding router gateway connected to the external network is selected as the start vertex of the graph traversal. Both of them use the same *prefix-to-prefix* function as `step 1`, except there are two *btrie* to encode prefix-level routing results in `step 2`.

4.3.3 VM-Level Isolation Verification

Prefix-level results computed in Section 4.3.2 are used to determine subnet pairs 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 4 describes the *VM-to-VM* procedure in which function $Route$-$Lookup$ checks whether there exists a forwarding path between any two VM ports, whereas the $VerifySecGroups$ function verifies security groups of these VMs.

Algorithm 4 *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-Lookup$(Triepub, Triepriv)$
4: if $routable = true$ then
5:     $VerifySecGroups(VM_{src}, VM_{dest})$
6: end if
```

The VM-to-VM route lookup is to get routing results of prefix-level reachability by searching in the relevant binary tries, while the destination IP addresses of these VM pairs are used as the key. As discussed in section 4.2, such query will find the boundary node of the most specific subrange including the individual VM IP. Therefore, the RLB value of the node can be retrieved as the routing result in virtual network graph for the VM pair. For example, considering the setting in example 8, and the x-fast trie shown in the step f of figure 4.7) as the prefix-level routing result, now we need to check the route from VM 10.1.0.2 to VM 10.2.0.15. The integer corresponding to 10.2.0.15 is used as key to query the trie. Then the second leaf from the left will be reached where RLB is 11, then we know routing from VM 10.1.0.2 to 10.2.0.15 is reachable. In addition, if VMs have public IP, two $btrie$ obtained from `step 2` of prefix-level verification are queried. Accordingly, the route lookups for these cases also consist of two substeps. Firstly, integer corresponding to the public IP of destination VM is used to query the x-fast trie obtained in `Step 2.a`, which is initialized with public prefix. Afterwards, the integer mapping to the private IP of destination VM is used as the key to query the trie generated in `Step 2.b`. Only RLB having value of 11 in both of query results means that a routing path passing external networks is present from the source VM to the destination VM.

Security Group	Direction	Prefix	Protocol	Port	Allowed
1	egress	10.2.0.0/28	tcp	3306	yes
1	ingress	10.2.0.0/28	tcp	3306	yes
2	egress	10.1.0.0/24	tcp	3306	yes
2	egress	any	tcp	80, 443	yes
2	ingress	10.1.0.0/24	tcp	3306	yes
2	ingress	any	tcp	80, 443	yes

Table 4.3: An Excerpt of Security Group Rules

Once a forwarding path is found between 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 tree and perform a first-match with the key from public or private IP of the destination VM. Then, we use the ingress security group of the destination VM and perform a first-match with the public or private IP of the source VM. 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.

We continue to use the setting in figure 4.2 with the sample security group rules shown in table 4.3. let say VM 10.1.0.1 and 10.2.0.15 are attached with security group 1 and 2 in table 4.3 respectively. Using integer mapping to the destination IP 10.2.0.15 as the key to query egress radix tree of security group 1, the filtering result will be allowed, since the key match the prefix 10.2.0.0/28. Similarly, it will have same outcome using integer mapping to 10.1.0.1 to query ingress radix tree of security group 2 attached to the destination VM. Therefore, the VM pair is verified to be reachable. If VM 10.2.0.15 is replaced with 10.2.0.16, then we get negative filtering result, since no match prefix can be found in security group 1. Hence, the two VM will be verified as isolated since the access is denied by the filtering rules associated with the source VM, even though a route exists between the pair.

We have shown how to verify isolation on VM-level. Although techniques such as radix trees are applied to improve scalability over security group rules, as well as prefix-to-prefix algorithm to address the repeated computation over routing, the amounts of VM pairs in large scale clouds are still huge. Consequently, the VM-level verification will be time-consuming for such clouds. We address this challenge by using parallel computation to distribute the verification tasks to multiple servers simultaneously, which will discussed in chapter 5.

4.3.4 Complexity Analysis

Let S be the number of subnets, R be the number of routers between two prefixes (i.e., number 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 3) 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 4) 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 the work by Ballani et al. [30] 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 $M' \ll M$.

Chapter 5

Application To OpenStack

We have implemented our baseline solution discussed in section 4.1, as well as TenantGuard based on prefix-to-prefix verification discussed in section 4.3, using OpenStack [8] as cloud management system, and Java as programming language. In this chapter, we describe our implementation details about data collection and preprocessing, parallel verification, and integration to OpenStack Congress.

5.1 Data Collection and Preprocessing

5.1.1 Data Collection

To build a snapshot of the virtual networking infrastructure for auditing, we collect data from OpenStack databases. In OpenStack, VMs are managed by the compute service `Nova`, while `Neutron` manages virtual networking resources in the cloud. Data related to these services are stored in two databases `nova` and `neutron`, while each of them contains over one hundred tables. Among them, information about VM ports, routers, subnets are located at table of `neutron` named `ports`, `routers` and `subnets` respectively, while some of their attributes are stored in other tables. For instance, assigned IPs for virtual ports are located in `ipallocations`. The relationships of virtual networking elements and these attributes are established by primary and foreign keys of these tables. Some of such mappings are listed in the table 5.1. For example, by primary

Primary Key	**Foreign Key**	**Established Relation**
nova.instances.ID	neutron.ports.device_id	VM and VM ports
neutron.ports.ID	neutron.ipallocations.port_id	Port and its private IP
neutron.ports.ID	neutron.floatingips.fixed_port_id	Port and its public IP
neutron.routers.ID	neutron.ports.device_id	Router and its interfaces
neutron.subnets.ID	neutron.ipallocations.subnet_id	Subnet and ports

Table 5.1: Some of Virtual Network Elements Data in `nova` and `neutron` Databases

Primary Key	**Foreign Key**	**Established Relation**
ports.ID	securitygroupportbindings.port_id	VM Port and its security group
routers.ID	routerroutes.router_id	Router and its forwarding rules
subnets.ID	subnetroutes.subnet_id	Subnet and its host routes

Table 5.2: Forwarding and Filtering Rules in OpenStack `neutron` Database

key `nova.instances.id` and foreign key `neutron.ports.device_id`, the connection between one VM and its attached virtual port can be identified. Similarly, by `neutron.ports.id` and `neutron.ipallocations`, IP addresses of ports can be collected.

On the other hand, we need to collect forwarding and filtering rules, and link them to corresponding virtual network elements. Table 5.2 shows primary and foreign keys in `neutron` database that establish relations between rules and elements. Note for routers, besides explicit rules directly extracted from the table `routerroutes` in `neutron` database, implicit rules associated with router interfaces are also fetched.

After sorting out above relationships, data with regard to virtual network elements and associated rules can be collected by SQL queries.

5.1.2 Data Preprocessing

Through data preprocessing, collected raw data from SQL queries are encoded into data structures discussed in Chapter 4, while the virtual networking graph is constructed accordingly.

Hash tables

From SQL query results, we instantiated various objects to represent virtual network elements, while sets of methods are developed to access attributes of these objects. Moreover, We use hash table to organize these objects, where UUIDs of virtual network elements are used as the keys. Therefore, to access any attribute or method of a given virtual network element, a search in hash tables using its UUID is performed. As a result, an object representing the targeting virtual network element is returned, and all methods related to that object are ready to be invoked.

Virtual Network Graph

The next step is to construct virtual network graph. JGraphT [31] is selected as the graph library in our application. The virtual network graph G inherits the simple graph in the JGraphT library. Vertices are inserted as discussed in Section 3.2, while edges are added in the way described in Table 3.1. Although vertex in JGraphT is type safe so that any java object could be used as a vertex, above virtual network element objects are not directly used as vertices. Instead, we create an unified vertex object where UUIDs are stored to keep the graph being compact. Hence, when a graph traversal arrives at one vertex and function fd_G need to be accessed, the corresponding object and related methods are available through UUIDs of objects stored in vertices.

Rules in Radix Trees

Forwarding and filtering rules are encoded into radix trees associated with objects of virtual network elements. For each router or subnet instance, there is one radix tree object containing all the associated rules. On the other hand, for each VM instance, there are two radix tree objects to encode ingress and egress filtering rules respectively. We implement our custom radix trees, offering query methods with various keys, as discussed in section 4.2.3, to support verification processes based on prefix-to-prefix algorithms.

In the raw data of SQL query result, nexthops of rules are represented as IP addresses, which could be overlapped between different tenant's private virtual networks. To offer features supporting overlapped IP addresses, in our application, UUIDs of corresponding virtual ports and routers are inserted to nexthops. Hence, for any given nexthop, it can be distinguished from other router interfaces using the same IP address.

X-Fast Binary Trie

In TenantGuard solution, one of the basic data structures is x-fast Binary trie. We implement it as introduced in Morin's book [32], where leaves are chained by a double list and one dumb node is introduced to join the both ends of leaves as a ring. Therefore, to traverse all the subrange in a x-fast trie, we can simply start from the dumb node along the double list in one direction, instead of repeatedly from the root node down to the leaves in the trie. Consequently, it is easier to efficiently implement algorithm 3 in which the iteration of all subranges in x-fast tries is needed. Similarly, such double list is used for range copy to traverse all leaves in that subrange, as explained in section 4.2.

On the other hand, in the original x-fast trie of Morin's book [32] the root node always links to nodes representing the first bit of integers. Whereas in our implementation, we use a prefix to initialize a trie, where the root node represents all bits of the prefix, since all subranges generated later will be within this prefix. For example, given the initial prefix 1001*, the root node represents integer corresponding to 1001. As a result, the level of tries reduces from l to $l - x$, where l is the number of bits for IP addresses, and x is length of prefixes in bits. There are twofold advantages from this modification: more compact space and less searching time, since one search costs $O(log(l - x))$ instead of $O(log(l))$. Additionally, we implement subrange write function in the x-fast trie in an atomic operation to ensure variables being consistent between a pair of boundary nodes.

Another implementation detail about x-fast trie is reusing temporary trie created during range copy. In our application, these temporary tries are cached into a hash table for reuse, where strings concatenating initial prefixes and router IDs are used as the keys. Hence, for a given router and a prefix, the query over this router's radix tree is only executed once during 3-step verification of TenantGuard.

With the virtual networking graph and preprocessed data, we implement our applications based on the baseline and TenantGuard solutions detailed in Chapter 4, which can run on a single verification server.

5.2 Parallel Verification

Parallelization Overview

As analyzed at the end of Section 4.3.3, the huge amount of VM pair in large-scale clouds will be a challenge, since ultimately we need to search routing results and verify security group rules for each VM pair in `step 3`, regardless to what extent we have improved the efficiency and scalability in previous steps. To address such challenge, we extend TenantGuard to a parallel environment in such a way that whole verification task could be divided into many pieces, which can be distributed and performed at multiple servers separately, and the outcomes can be collected and summarized immediately when they are available. Figure 5.1 demonstrate the architecture of such parallelization.

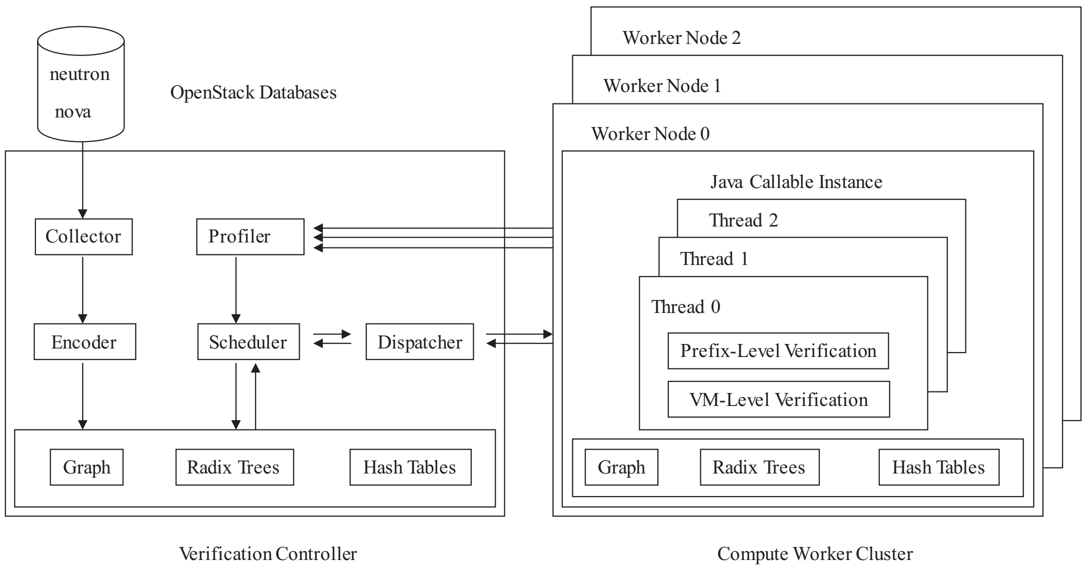


Figure 5.1: TenantGuard Parallel Implementation

We use open source Apache Ignite [33] as our parallel computation platform, enabling to distribute workload in real time across hundreds of servers. TenantGuard parallelization consists of two parts, the verification controller and the compute worker cluster.

Verification Controller

The verification controller runs TenantGuard parallelization application on a single server with Ignite platform. The parallel version of TenantGuard consists of following modules:

- `Collector`
- `Encoder`
- `Profiler`
- `Scheduler`
- `Dispatcher`

`Collector` and `Encoder` are the modules to perform data collection and preprocessing, as discussed in Section 5.1. `Profiler` collects the computation capacity and metrics of the compute worker cluster, such as the number of cores, CPU loads, etc. According to the data generated by `Profiler`, `Scheduler` dynamically divides the verification task on the prefix level and encapsulate the pieces as Java callables, while `Dispatcher` works in an asynchronous mode to distribute these Java callables and the data such as graph, radix trees and hash tables to compute worker nodes through data streaming, where java callables are executed individually follow the three-step verification process discussed in Section 4.3. To support object serialization in data streaming, `Serializable` interface is implemented in all involved Java classes of TenantGuard. When each distributed task terminates, the result is collected immediately by `Dispatcher` and then summarized by `Scheduler`.

Compute Worker Cluster

The cluster consists of the Ignite nodes, which discover each other either automatically when they are located in the same LAN, or through ignite configuration files when they are distributed in a more complex networking environment. For each node, only JVM and Ignite platform are needed to be deployed. All the code and data for TenantGuand verification are received in runtime on demand. When tasks are received, the three-step verification codes and data are extracted from the java callables. Local resources such as number of CPUs are obtained through Java `Runtime` instance and used to calculate appropriate number of threads. Subsequently, verification tasks are further divided into multiple threads that are performed simultaneously so that local resources can be fully utilized. In each thread, when prefix-level and VM-level verification are finished, the result is reported to the callable. After all threads exit, the overall result is sent back to the controller.

With TenantGuard parallelization, we can easily adjust the number of compute workers in the cluster, to handle VM -level virtual network isolation verification for clouds with different scales, since each worker only needs JVM and Ignite installed. Hence, TenantGuard is scalable to large-scale clouds verification.

5.3 Integration to OpenStack Congress

We further integrate TenantGuard into OpenStack Congress service [25]. 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 [25]. 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 Chapter 2). 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.

Chapter 6

Experiments

This section presents experimental results. At first, we introduce the settings of our experiments, including data sets and testing environments. Secondly, we report a set of results that quantitatively compare TenantGuard not only with our baseline solution, but also with NoD [21] and its later work [7]. Finally, we show the results that our application runs on a real cloud.

6.1 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 with OpenStack [8]. There are one controller node integrated with networking service, and up to 80 compute nodes. Tenants' VMs are initiated from the Tiny CirrOS image [8], separated by VLAN inside the compute nodes, while VxLAN tunnels are used for the VM communication across the compute nodes.

We generate two series of datasets (i.e., SNET and LNET) for the evaluation. The SNET dataset represents small to medium virtual networks containing six subnets, routers and VMs generated in OpenStack, 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 correspondent characteristics of our algorithms. On the other hand, the LNET dataset represents large networks, where each virtual network is organized in a three-tier structure in such a way that one first-tier router is connected

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 6.1: LNET Dataset Description

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 other tenants, 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 [8], 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 6.1. We use open-source Apache Ignite [33] as the parallel computation platform, which can distribute the workload in real-time across hundreds of servers. On the other hand, all datasets for NoD are generated synthetically using the provided generator[1].

6.2 Results

We evaluate the performance of our approaches and the effect of various factors on the performance, as well as abilities to identify networking anomaly such as forwarding loops .

6.2.1 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 CPU and 2GB memory. Note that in this part of experiments, the verification time for NoD is measured for 1 to 5 pairs, and for TenantGuard the time is measured for all-pair.

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

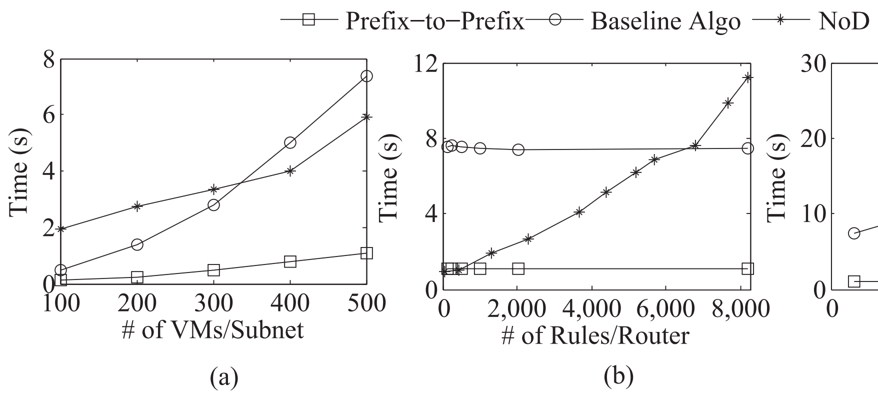


Figure 6.1: SNET Performance Comparison

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

The first examined factor is VM density in subnets. As shown in Figure 6.1(a), when the number of VMs per subnet is increased from 100 to 500, the prefix-to-prefix isolation verification time increases much slower than the baseline algorithm (defined in Section 4.1) and Nod, which shows TenantGuard's better scalability. The reason behind these results is, as illustrated in complexity analysis in section 4.3.4, the prefix-to-prefix algorithm 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 $O(R + N^2)$.

Another factor is the number of forwarding rules in routers. As shown in Figure 6.1(b), when the number of routing rules per router increase exponentially, the verification time remains relatively stable due to using radix tree and X-fast trie, both of which have constant searching time. Still, the performance of prefix-to-prefix algorithm is only nearly one quarter of the baseline's for each setting, which shows the improved efficiency of TenantGuard. On the other hand, NoD verification time mainly depends on the number of rules, hence, increasing the number of rules results in a significant increase in the verification time.

The last factor is hops between VMs. As it 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

[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

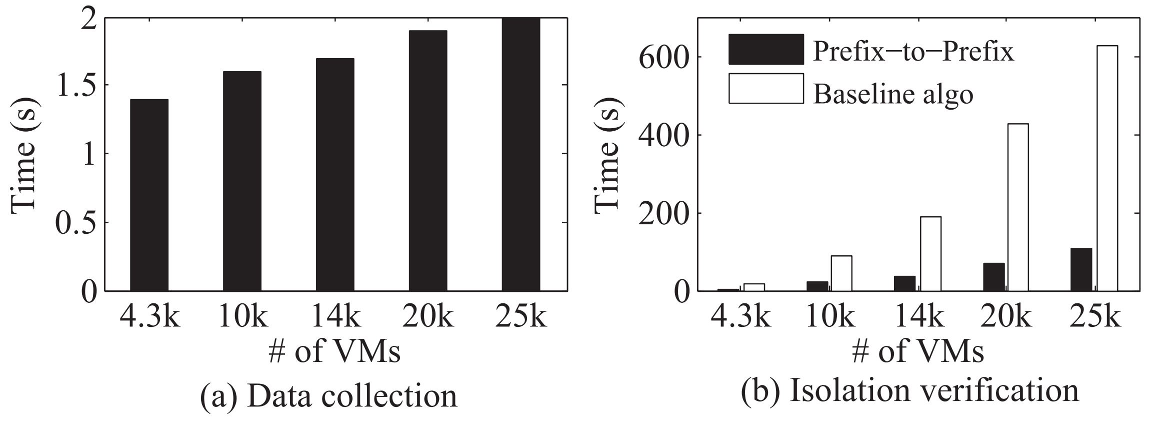


Figure 6.2: LNET Performance Comparison in the Single-Machine Mode

(a)Showing Data Collection Time, and (b) Showing Verification Time

(e.g., Internet) and according to related researches [34] [35], the average number of hops varies between 12 to 19; hence, to stress out our approach more, we vary the number of hops between 2 to 26. Figure 6.1(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. We exclude NoD from this experiment, as the NoD algorithm does not consider hops as one of the parameters.

Overall, TenantGuard is more efficient and scalable over all of the three factors, comparing with the baseline algorithm and Nod.

6.2.2 LNET Results Using Amazon EC2

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, routers and total amount of VMs are varied as shown in Table 6.1. There are two modes for *LNET* tests: single-machine mode and parallel mode.

Single-Machine Mode

Single-machine tests are conducted on one EC2 C4.large instance at AWS EC2 with 2 vCPU and 3.75 GB memory. As shown in Figure 6.2(a), data collection and processing time varies between 1.5 to 2 seconds, including data collecting and preprocessing, which shows that the collection time is not the prominent part of the execution time. Meanwhile, Figure 6.2(b) compares the verification

NoD	Nod with symmetry and surgery	TenantGuard
1000 routers, 820k rules, 100k VMs	Same as NoD	1200 routers, 850k rules, 100k VMs
131 hours	2 hours	1056 seconds

Table 6.2: Comparison of All-pair Reachability with NoD

time between the the baseline and TenantGuard. When the number of subnets and VMs increases, the prefix-to-prefix algorithm is more efficient than the baseline algorithm. For 25,246 VMs in 3,250 subnets, it takes 108 seconds using the prefix-to-prefix algorithm and around 628 seconds for the baseline algorithm. Note that totally over 168 millions VM pairs are verified as reachable in this dataset.

Comparison with NoD

In Plotkin et al. [7], all-pair VM reachability verification results are reported over one data center with less than 1000 routers, 820k forwarding and ACL rules, and 100k VMs. To compare with them, we increase number of VM to 100k as well as rules to 850k based on our dataset DS5, which has 1200 virtual routers. TenantGuard is tested over this extended dataset, and the comparison is reported in table 6.2. NoD takes 131 hours(about 5.5 days) to verify all-pair reachability while with symmetry and surgery in its later work [7] it is improved to 2 hours, while TenantGuard only takes approximately 18 minutes to verify all-pair reachability over a dataset with similar scale. Note our comparison is limited on all-pair reachability, since NoD is designed as a general networking verification tool.

Parallel Verification Test

Although our approach already demonstrates significant performance improvements over the baseline algorithm, the results are still based on a single machine. In a real cloud, the verification tasks can be easily parallelized among many machines to further improve the performance. 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

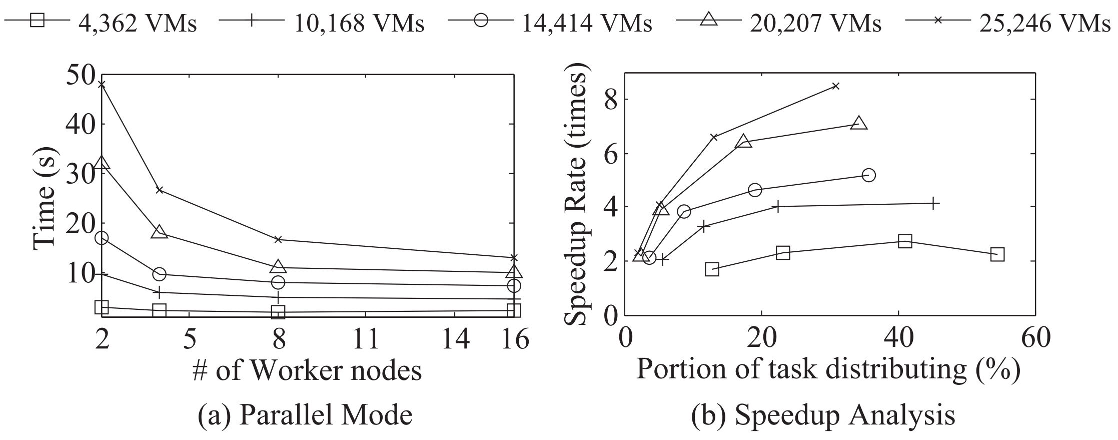


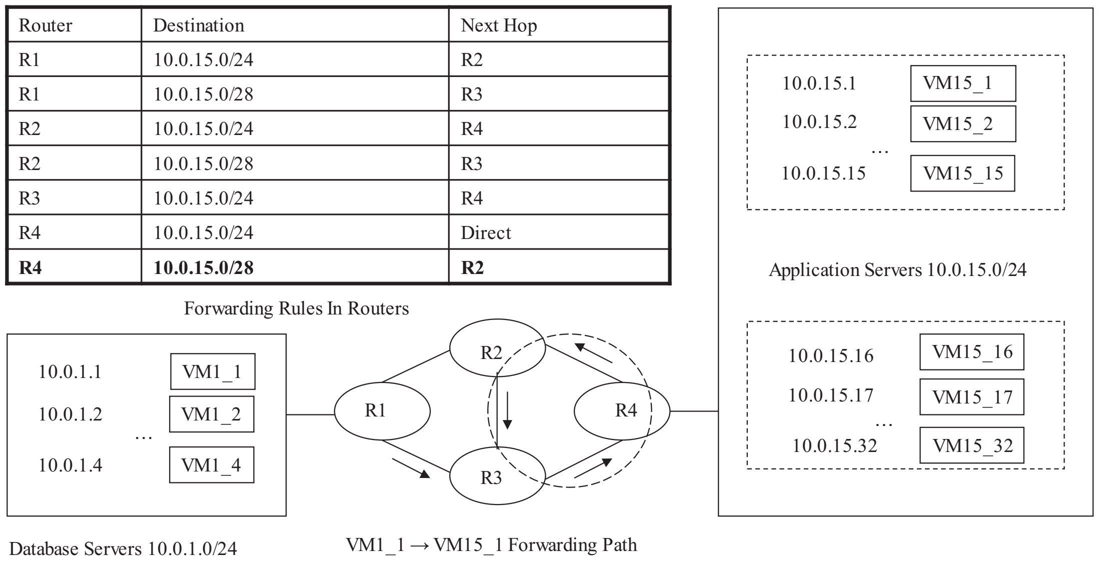
Figure 6.3: Performance Improvement of Parallel Verification with LNET Data

(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

with 4 vCPU 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 6.3(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 6.3(b), we examine the relationship between the cluster size and speedup gain. The overhead of parallel computation 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 time. 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



Router	Destination	Next Hop
R1	10.0.15.0/24	R2
R1	10.0.15.0/28	R3
R2	10.0.15.0/24	R4
R2	10.0.15.0/28	R3
R3	10.0.15.0/24	R4
R4	10.0.15.0/24	Direct
R4	**10.0.15.0/28**	**R2**

Figure 6.4: Forwarding Loop Detection with Injected Rules

an asynchronous task distribution technique. In Figure 6.3(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 6.3(b)) illustrates that for the smallest data set DS1, after 8 nodes the speedup goes down. Therefore, we restrict the number of working nodes to 16.

Forwarding Loop Detection

We also test forwarding loop detection by injecting error data, as shown in figure 6.4. The setting is similar to figure 4.2, but with some configuration changes. Firstly, a link is added between router R2 and R3, while IP addresses follow what we use in experiments. Secondly, a rule is added in R2 for redirecting packets to 10.0.15.0/28 towards R3. Finally, we inject a rule in R4 which causes loop in forwarding paths from database servers to application servers, which is in bold at the last row of the upper table in figure 6.4. Consider a forwarding path from `VM1_1` to `VM15_15`, which is labeled with arrows in the figure. In R1 the packet will be forwarded to R3, since the most specific prefix for the destination 10.0.15.15 in R1 is 10.0.15.0/28. Sequentially, it will be forwarded to R4, then the injected rule in R4 will route packets to R2, then come back to R3, where a forwarding loop is formed. On the other hand, from the same source, packets to `VM15_16` which IP address is 10.0.15.16 will be routed correctly, due to the fact that the most specific matching prefix for 10.0.15.16 is 10.0.15.0/24 in R1, R2, and R4. In the experiments, our tool can accurately identify every VM pair with forwarding loops such as from `VM1_1` to `VM15_15`, as well as normal forwarding path such as from `VM1_1` to `VM15_16`, although the two destination VMs are located in the same subnet.

6.2.3 Experiment with Real Cloud

We further tested 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 incompatability 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.

Chapter 7

Discussion

In this discussion, 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.

Routing and Filtering	**OpenStack [8]**	**Amazon EC2-VPC [36]**	**Google GCE [37]**	**Microsoft Azure [38]**	**VMware vCD [39]**	**TenantGuard Support**
Intra-tenant routing	Host routes, routers	Route tables	Routes	System and user-defined routes	Distributed logical routers	Yes/ TenantGuard forwarding and filtering function
Inter-tenant routing	Routers, external gateways	Internet gateway and 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 7.1: TenantGuard Supported Routing and Filtering in Different Cloud Platforms

We review packet routing and filtering in different cloud platforms and show the applicability of TenantGuard. Table 7.1 shows how routing and filtering are implemented in OpenStack, Amazon AWS EC2-VPC (Virtual Private Cloud) [36], Google Compute Engine(GCE) [37], Microsoft Azure [38], and VMware vCloud Director (vCD) [39]. 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 are existing techniques on trusted auditing to establish a chain of trust, e.g., [40, 41, 42]. Bellare et al. [40] propose a MAC-based approach. However, they provide the forward integrity by using chain of keys and erasing previous keys so that any old logs cannot be altered. Crosby et al. [41] 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. [43] implements 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 [44].

Chapter 8

Conclusion

In this thesis, 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.

Limitations and Future Works.. The main limitations 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.

Bibliography

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

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

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

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

[5] openstack.org. OpenStack user survey, October 2015. Available at: https://www.openstack.org.

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

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

[8] OpenStack. OpenStack open source cloud computing software, 2016. Available at: http://www.openstack.org.

[9] OpenStack Security Advisory. 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.

[10] OpenStack Security Advisory. 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.

[11] J. Corbet. Trees I: Radix tree, 2006. Available at: http://lwn.net/Articles/175432/.

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

[13] 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.

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

[15] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. Network configuration in a box: towards end-to-end verification of network reachability and security. In *ICNP*, 2009.

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

[17] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, pages 1–11, Oct 2013.

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

[19] 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. 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.

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

[22] S. Bleikertz, M. Schunter, C. W Probst, D. Pendarakis, and K. Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *CCSW*, 2010.

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

[24] 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.

[25] OpenStack Congress. Congress documentation release, 2016. Available at: https://congress.readthedocs.io/en/latest/.

[26] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[28] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

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

[30] 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.

[31] Barak Naveh. Jgrapht, 2014. Available at: http://jgrapht.org/.

[32] Pat Morin. *Open Data Structures: An Introduction*. Edmonton, AB : AU Press, [2013] 2013, 2013.

[33] Apache Software Foundation. Ignite, 2015. Available at: https://ignite.apache.org.

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

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

[36] Amazon. Amazon virtual private cloud, 2015. Available at: https://aws.amazon.com/vpc.

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

[38] Microsoft. Microsoft azure virtual network, 2015. Available at: https://azure.microsoft.com.

[39] VMWare. Vmware vcloud director, 2015. Available at: https://www.vmware.com.

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

[41] Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

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

[43] 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*, pages 73–84. Springer, 2003.

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