VMGuard: State-based Proactive Verification of Virtual Network Isolation with Application to NFV

Gagandeep Singh Chawla, Mengyuan Zhang, Suryadipta Majumdar,
Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi

Abstract—Network Functions Virtualization (NFV) leverages from clouds to simplify and automate the creation and deployment of network services on the fly in a multi-tenant environment. However, clouds may also bring issues leading to tenants' concerns over possible breaches violating the isolation of their deployments. Verifying such network isolation breaches in cloud-enabled NFV environments faces unique challenges. The fine-grained and distributed network access control (e.g., per-function security group rules), which is typical to virtual cloud infrastructures, requires examining not only the events but also the states of all virtual resources using a state-based verification approach. However, verifying the state of a virtual infrastructure may become highly complex and non-scalable due to its sheer size paired with the self-serviced dynamic nature of clouds. In this paper, we propose VMGuard, a state-based proactive approach for efficiently verifying large-scale virtual infrastructures in cloud and NFV against network isolation policies. Informally, our key idea is to proactively trigger the verification based on predicted events and their simulated impact upon the current state, such that we can have the best of both worlds, i.e., the efficiency of a proactive approach and the effectiveness of state-based verification. We implement and evaluate VMGuard based on OpenStack, and our experiments with both real and synthetic data demonstrate the performance and efficiency, e.g., less than five milliseconds to perform incremental verification on a dataset with more than 25,000 VMs and less than two milliseconds with the proactive module enabled.

Index Terms—Security Compliance, Verification, Cloud Security, Security Auditing, Network Isolation

1 Introduction

Security and privacy issues, such as lack of transparency, accountability and verifiability, remain the main concerns for individuals and companies when it comes to adopting clouds [1], [2], [3], [4] and its related technologies, particularly network functions virtualization (NFV) [5], which is an emerging cloud-enabled application that allows the virtualization of network functions and the delivery of network services using cloud networks. While NFV benefits from the promising features of the cloud service delivery model such as cost optimization, fast deployment and scalability, it inherits several challenges such as the lack of visibility into the underlying infrastructure, which hinders its auditability [4], [6].

In particular, the multi-tenancy nature of public clouds means cloud tenants would likely keep worrying about the lack of sufficient network isolation around their virtual resources. Recent studies show that almost 70% of cloud users consider security as a major issue in clouds, of which 80% agree that network isolation is the biggest obstacle to adopting clouds [7], [8]. Cloud providers often have an obligation to provide clear evidences for sufficient network isolation [9], either as part of the service-level agreements, or to demonstrate compliance with security standards (e.g., ISO 27002/27017 [10], [11] and CCM 3.0.1 [12]). Moreover, cloud providers may gain a competitive edge in today's market by providing the capability of verifying network isolation as a security service to their tenants.

However, in contrast to traditional network environments, the virtual infrastructures hosted in clouds pose unique challenges to network isolation verification. The sheer size of such virtual infrastructures paired with their self-serviced and highly dynamic nature means most verification techniques designed for traditional networks would cause so much delays that their results may become obsolete before they are ready (a more detailed review of related work is provided in Section 7). To that end, two promising cloud-specific approaches are: state-based verification and proactive verification. The state-based verification approach verifies the entire virtual network states (e.g., reachability information inside a virtual infrastructure), and hence, achieves a high accuracy rate. However, due to the sheer size of a cloud, the existing works (e.g., [13], [14]) may cause significant delay. On the other hand, the proactive verification approach (e.g., [15], [16]) responds very quickly, as it adopts an event-based approach and initiates the verification before an event actually occurs. However, examining only the events and overlooking the entire state of a network could affect its accuracy; this will be further explained in Section 2 to demonstrate the strengths and weaknesses of those solutions and motivate towards our work.

Considering the above-mentioned advantages and drawbacks of the state-based approach and the proactive approach, a natural question is: *Can we design a state-based, and proactive approach, such that we can have the best of both worlds?* In this paper, we propose *VMGuard* as an answer.

- *G. S. Chawla, S. Majumdar, L. Wang and M. Debbabi are with The Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada. E-mail: g_chawla, majumdar, wang and debbabi@encs.concordia.ca*
- *M. Zhang, Y. Jarraya and M. Pourzandi are with The Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada. E-mail: mengyuan.zhang, yosr.jarraya and makan.pourzandi@ericsson.com*

We tackle several unique challenges in designing VMGuard. Specifically, the state-based verification only works after an event has actually occurred (with its effect on the state materialized). However, since each event can lead to multiple predicted next events with different effects on the current state, how to verify those predicted events without adversely affecting the true state of the virtual infrastructure (since predicted events may never happen) becomes a major challenge. We will tackle this and other challenges in the remainder of this paper. In summary, our main contributions are as follows.

- To the best of our knowledge, VMGuard is the first state-based proactive approach to network isolation verification. Its novel approach first proactively performs verification on the predicted next events with all potential parameters, then incrementally updates the verification results with the evolution of the virtual infrastructure, and finally enforces its verification result on the infrastructure as soon as an actual event occurs.
- Our experiments with both real and synthetic data demonstrate both effectiveness and efficiency of VMGuard in comparison to existing methods as demonstrated through experiments with both real and synthetic data. As VMGuard is proactive, it takes less than two milliseconds for a dataset with more than 25,000 VMs for any intercepted event. In contrast, TenantGuard verifies all pairs reachability and takes 108 seconds for the same dataset.
- We implement VMGuard based on OpenStack [17], a major cloud management platform. Additionally, we provide thorough discussion on how to port VMGuard to the NFV environments.

The remainder of the paper is organized as follows. Section 2 briefly describes the preliminaries of this work. Section 3 details the methodology. Section 4 presents the implementation and Section 5 shows the experimental results. Section 6 provides more discussions. Section 7 reviews related work and Section 8 concludes the paper.

2 PRELIMINARIES

This section first provides necessary background on two state-of-the-art verification approaches to facilitate more discussions, then further motivates towards our solution using an example, and state our threat model. Finally, we brief over the NFV reference architecture and network service deployment.

2.1 State-Based Verification

As an example of state-based (the *state* here refers to the collection of all reachability information inside the virtual infrastructure) verification tools, TenantGuard [14] can efficiently verify all-pair VM-level reachability for large clouds (e.g., 13 seconds for 168 millions of VM pairs) with its hierarchical approach, as demonstrated in Figure 1 and detailed below.

- In `[Step 1]`, TenantGuard checks the subnet-to-subnet reachability within the same network (using the private IPs), e.g., between SN_{A1} and SN_{A2}.

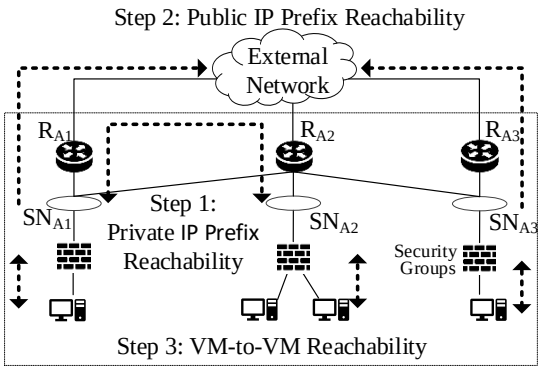


Fig. 1: Examples of the state-based verification by TenantGuard [14]

- In `[Step 2]`, TenantGuard verifies the subnet-to-subnet reachability involving external networks (using the public IPs), e.g., between SN_{A1} and SN_{A3}.
- In `[Step 3]`, TenantGuard only needs to check VM-level reachability (mainly based on security group rules of VMs) for the reachable subnets obtained from the first two steps, leading to significant cost savings.

2.2 Proactive Verification

As an example of proactive verification tools, LeaPS [16] initiates the verification before an event actually arrives, leading to millisecond-level response time. This is possible due to the so-called dependency model, which captures the likelihood of next events (either by design, or extracted as frequent patterns from historical events), as demonstrated in Figure 2 and detailed below.

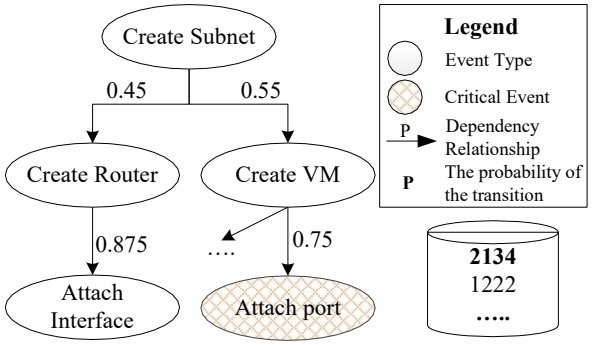


Fig. 2: Examples of the proactive verification by LeaPS [16]

- When a *create VM* event occurs, LeaPS identifies a highly probable next event, *attach port*, from the dependency model.
- LeaPS verifies this predicted event *(attach port)* based on a pre-defined signature of critical events (i.e., events causing an isolation breach between different cloud users/tenants, e.g., plugging a port on another tenant's VM under vulnerability OSSA 2014-008 [18]), and if this event causes no isolation breach then its parameters (e.g., VM ID: *2134*) will be added to the watchlist (which is essentially a tenant-specific white list).

- Later, when an *attach port* event actually occurs, all LeaPS needs to do is to search for its parameter (e.g., VM ID) in the watchlist (a match means the event will be allowed), resulting in a negligible delay.

Using the following example, we demonstrate the drawbacks and strengths of those exisitng approaches, and position our solution, VMGuard.

2.3 Motivating Example

Figure 3 employs a sequence of administrative events (upper-left) inside a simplified virtual infrastructure (upper-right) to demonstrate the limitations of existing approaches and our key ideas (lower). Suppose a user *Bob* wants to forbid all the ingress traffic from subnet SN_{A1} to his billing server (IP: 10.1.1.7) by deleting its security group rule [17]. However, he is not aware of an OpenStack vulnerability OSSA-2015-021 [19], which causes the change of security group to fail silently on already running VMs. At a later time, another user *Alice* creates her own VM (10.1.5.9) and connects it to subnet SN_{A1}. At this time, the network isolation has been compromised since *Alice* can now access *Bob's* billing server via subnet SN_{A1}.

- In such a case, the state-based verification approach could generate all-pair reachability results within seconds by examining the state of the virtual infrastructure. However, the current solutions (e.g., [13], [14]) by themselves do not support verification against isolation policies (e.g., no ingress traffic from subnet SN_{A1} to Bob's billing server), and applying an additional verification tool (e.g., [20]) on top of the all-pair reachability results may again introduce a significant delay (ranging from minutes to hours for verifying large clouds based on our evaluation).
- As a promising solution, proactive verification (e.g., [15], [16]) would predict event E_K as soon as it sees E_{K-1} since the next operation after creating a VM is typically attaching it to a subnet), and consequently start the verification for E_K long before it actually arrives. However, since we assume the vulnerability is not known, the first half of this undesired reachability (from Bob's billing server to SN_{A1}) can only be detected by examining the state of the virtual infrastructure. Therefore, without looking at the state, the event-based proactive verification will consider E_K as normal and miss the isolation breach of *Bob*'s VM (i.e., *Alice* VM can still reach to *Bob*'s.

In this paper, we propose *VMGuard* as an answer, as depicted in the bottom of Figure 3. We elaborate on our methodology in Section 3.

2.4 Threat Model

The in-scope threats include any implementation flaws, misconfigurations, and certain vulnerabilities in the cloud platform that may be exploited by malicious cloud users that lead to a change of the system state. If the change could potentially violate the network isolation policies specified by cloud tenants or the provider, VMGuard would be able to block the event. Since our solution is based on the system state instead of signature, such threats are in the scope even if they are previously unknown, as long as their effect on network isolation is visible in the state of the virtual infrastructure. On the other hand, we focus on the virtual network management layer in the cloud and only consider network isolation-related security policies. We also assume the cloud platform may be trusted for the correctness of the inputs (e.g., logs and configuration database). Any security breach that is not reflected in those inputs (either due to the nature of such breaches, such as side-channel attacks, or due to inputs tempered by attackers) is out of the scope. Also, any potential privacy leakage from the verification results is beyond the scope of this work. The pruning step in VMGuard design is optional and should be applied by a user if s/he is only concerned by the vulnerabilities that affect a specific part of the cloud.

In-Scope Misconfigurations and Vulnerabilities. The possible misconfigurations include the reachability created by admins accidentally. For example, an admin may add a VM from one tenant to another or delete an important VM from one tenant mistakenly. These behaviors may not be malicious by nature and hence, an intrusion detection solution may not raise any alarm. VMGuard could block those potential misconfigurations by verifying against tenant-defined policies. The unknown vulnerabilities that might breach network isolation between tenants could be monitored by VMGuard against the policies. For example, if an unknown vulnerability allows one tenant to stealthily add his/her VM under another tenant's subnet, signature-based security solutions may fail to capture this as the signature remains unknown while VMGuard can still protects the system against such unwanted information leakage based on tenant-defined policies. Generally, even the attack steps are unknown or can be improvised in many ways by an attacker, VMGuard can still detect those attacks as long as they breach a given security policy related to network isolation.

2.5 NFV over Clouds

To enable the deployment of NFV network services, ETSI [5] standardizes the high-level NFV reference architecture. NFV resides as a logical layer over clouds to automate, orchestrate and manage network services.

NFV Reference Architecture. Figure 4 illustrates the ETSI NFV reference architecture consisting of the Management and Orchestration (MANO) with its three major components as the driving unit for NFV. The Virtual Infrastructure Manager (VIM), manages and controls Network Function Virtualization Infrastructure (NFVI) i.e., the pool of virtualized compute, storage and network resources. The Commercial Off-The-Shelf (COTS) servers are combined to act as a common pool of resources, which are then logically redistributed either in presence of a hypervisor, i.e., Virtualization, or in absence of it, i.e., Containerization. Note that, VMGuard is designed to work with the clouds, therefore, in our use-case, we only cover virtualization-based VIM i.e., OpenStack. Over NFVI resides the Virtual Network Functions (VNFs) that undergo common health operations such as scale, heal, terminate, etc. as a part of their life-cycle [21]. A Virtual Network Function Manager (VNFM) manages the life-cycle of VNFs. Some VNFs exhibit specific health-checks and mission-critical operations.

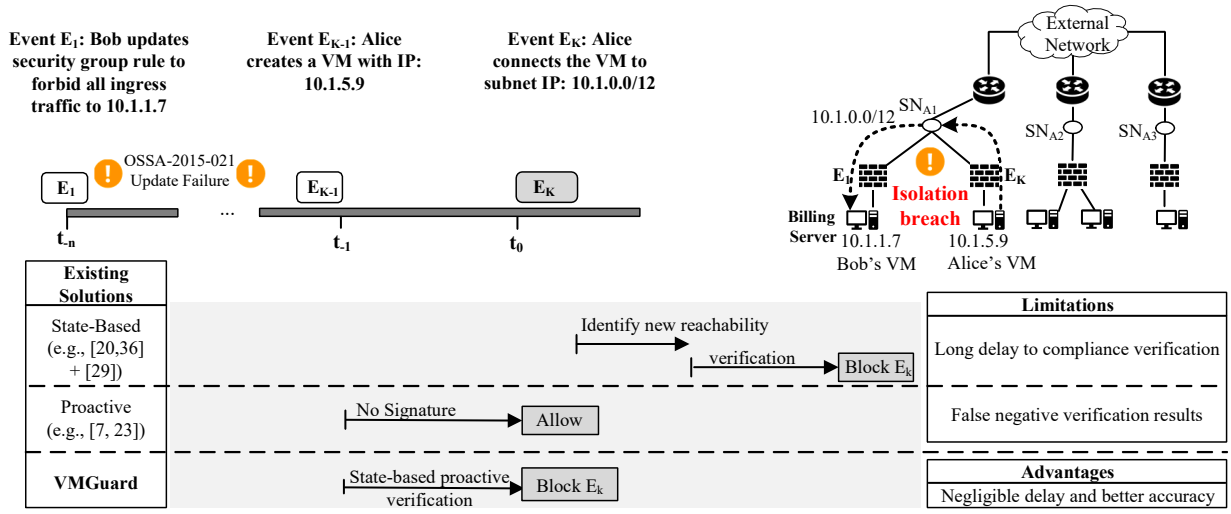


Fig. 3: The motivating example

Therefore, they receive a dedicated life-cycle manager i.e., Element Management Systems (EMS), facilitated by their vendors. VNFM also coordinates with the EMS. The VNFs are launched as a part of network services by the Network Function Virtualization Orchestrator (NFVO). The latter acts as a management interface, where the client specifies their network service requirements. Finally, the Business/Operations Support System (OSS/BSS) supports end-to-end telecommunication services requirement gathering, which is encoded as the descriptors [22].

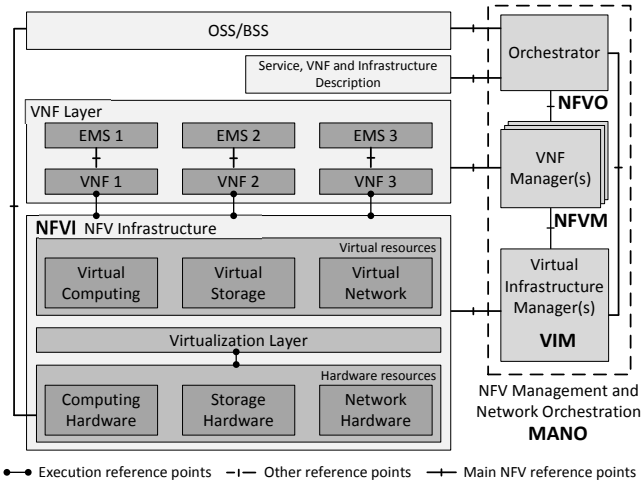


Fig. 4: ETSI NFV architecture [5]

Deploying a Network Service. The OASIS[1] standardizes descriptors as the means to encode interoperable services and application workloads hosted on the clouds [23]. The TOSCA [24] descriptors enable interoperability and portability with automated management across the cloud platforms [22]. Three types of descriptors are used during a network service deployment process. A VNF Descriptor (VNFD) describes the specification of a VNF. Several VNFs can be connected with the forwarding paths (i.e., SDN-based), described as a VNF Forwarding Graph Descriptor (VNFFGD). The forwarding graphs require Neutron-SFC (Service Function Chaining) [22], which we do not support currently with VMGuard. A Network Service Descriptor (NSD) describes the complete end-to-end network service, composed by connecting VNFs and the forwarding graphs.

Assuming Bob in the motivating example as a NFV tenants (an NFV tenant is not necessarily a cloud tenant, as elaborated in Section 6). Bob interacts with the OSS/BSS demonstrating his requirements and generates a TOSCA for his/her network service. TOSCA is provided to the NFVO, which uses a TOSCA-Parser [22] to produce a Heat template. Heat [22] is an OpenStack [17] orchestration engine that enables the cloud tenants to automate the creation and deployment of the virtual resources as workloads written as Heat Orchestration Template (HOT) [25]. The templates are human and machine readable codes written in YAML which manages cloud resources. The heat template generated from TOSCA defines all the virtual resources such as VMs, ports, routers, required for Bob's network service. The heat template is received by OpenStack (VIM) to orchestrate the network service in the cloud. NFVO co-ordinations the deployment of virtual resources and acknowledges Bob with the deployment status. VMGuard resides at VIM, where it analyzes heat templates to first propose policies for the tenants and then to enforce the applied ones, as elaborated later in Section 3.6.

3 VMGuard Methodology

We first show the main challenges and provide an overview before delving into the details of our methodology.

3.1 Challenges

While previously discussed state-based verification (e.g., TenantGuard [14] in Section 2.1) and proactive verification (e.g., LeaPS [16] in Section 2.2) can each work efficiently on the aspects it was designed for, they cannot simply

1. Organization for the Advancement of Structured Information

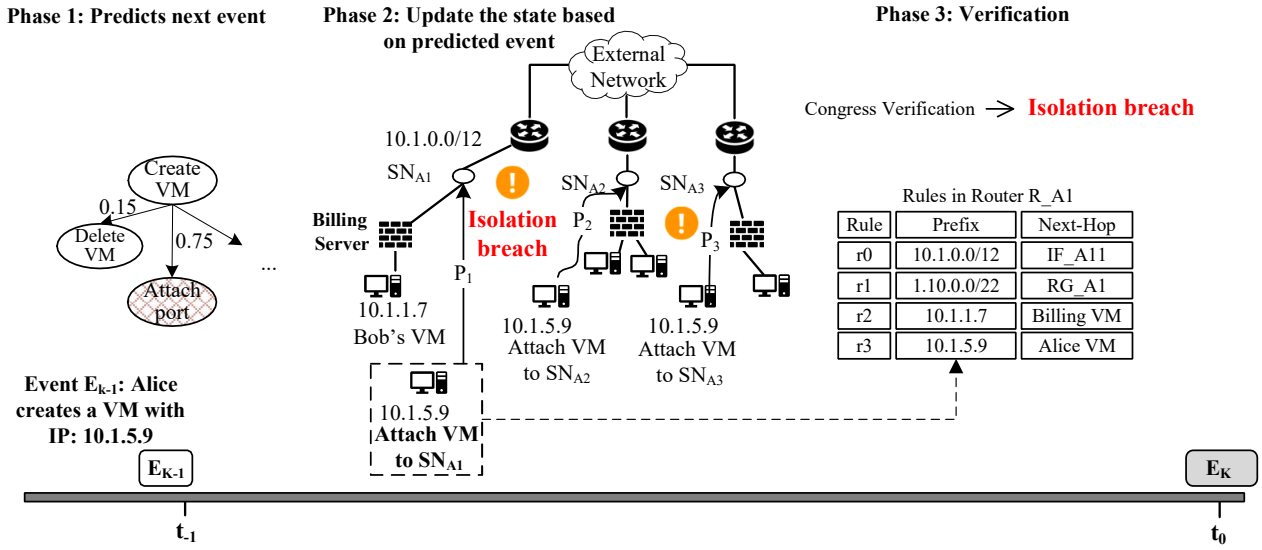


Fig. 5: Challenges in designing a state-based proactive verification approach

be orchestrated to enable scalable and efficient state-based proactive verification, without fundamental modifications to their inner working. In this section, we discuss the main challenges encountered in this respect based on the integration scenario as illustrated in Figure 5. Then, we show how we address them in our solution.

In Figure 5, in `[Phase 1]`, upon the occurrence of event E_{k-1} (*create VM*), LeaPS is used to predict the next event to be *attach port* based on the dependency model. In `[Phase 2]`, we need to obtain an updated state of the virtual infrastructure resulting from applying the effect of this predicted event on the system. Having the states of the system (i.e., the collection of all the reachability information with the effect of that event taken into consideration) is essential to applying any state-based verification such as TenantGuard. In `[Phase 3]`, TenantGuard is applied to the updated state to obtain the reachability result. In this case, TenantGuard identifies a breach of isolation between Alice and Bob networks, the watchlist (white list) is not updated, and when the actual event E_k (*attach port*) arrives, it will be denied. However, this description has omitted some major challenges related to obtaining such a state for an event that is only predicted, as follows:

- **(C1) Lack of mechanism to predict events' parameters**: The dependency model can only predict the type of events (e.g., *attach port*) but not their detailed parameters (e.g., to which subnet) [16]. However, without the right parameters, we cannot obtain the state resulting from the effect of a given event. To solve this issue, we propose a pre-computation approach for these parameters (see Section 3.3 for more details).
- **(C2) Complexity of obtaining the actual resulting state**: even if we could predict the event parameters (e.g., *attach port* to either subnet SN_{A1}, SN_{A2}, or SN_{A3}), obtaining the resultant state for those events would still be a challenge. We cannot directly apply such events to the virtual infrastructure, since the predicted events may never occur, while their effects may be irrevocable (e.g., information leakage or denial of service as the result of an isolation breach). We address this challenge by evaluating the possible states using incremental verification (see Section 3.4 for more details).
- **(C3) Prohibitive Computational and Storage Overhead**: While a viable solution is to *emulate* the effect of predicted events on a new copy of the current state, however, creating such a copy may lead to prohibitive computational and storage overhead in clouds. To make things worse, an event may lead to multiple predicted parameters with different and incompatible effects on the state (e.g., P_1 and P_3 lead to breaches but P_2 does not), which requires creating multiple independent copies of the state. We use a pruning technique to tackle this issue (see Section 3.5 for more details).

3.2 Overview

To address the aforementioned challenges, we design our state-based proactive approach, namely, *VMGuard*. Figure 6 shows an overview of VMGuard with three main modules, proactive verification, incremental verification, and pruning.

- First, the proactive verification module performs pre-computation and proactive verification. In the pre-computation phase, all the predicted next events will be instantiated with potential parameters based on the current state of the virtual infrastructure.
- Second, the incremental verification module forks multiple copies of the state each of which is used to evaluate the impact of one instantiated event. Each copy of the state is flushed immediately once the verification is done; only the compliant results are stored in a watchlist. In this context, compliant means the actual implementation meets the user's specifications (e.g., security policies), while non-compliant means the implementation violates the specifications.

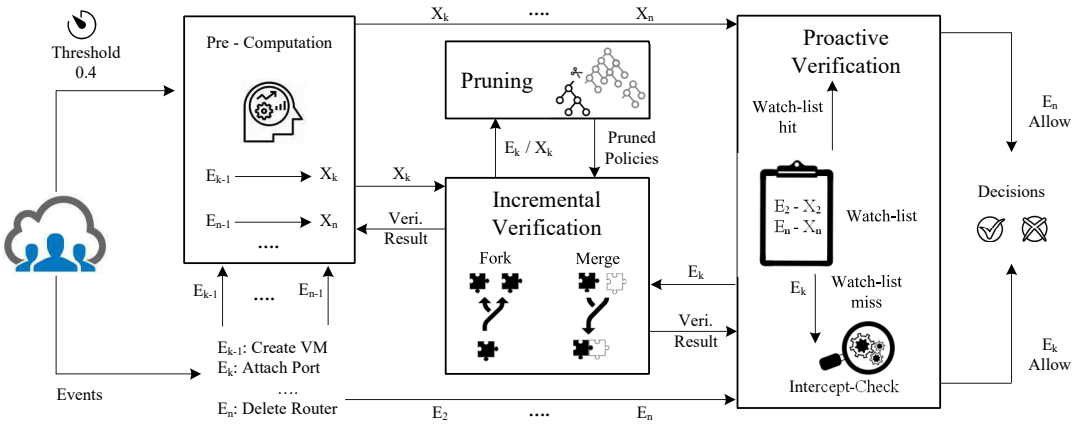


Fig. 6: An overview of VMGuard

- Third, to minimize the overhead of incremental verification, the pruning module further limits the scope of the verification to the intersection between the set of VMs involved in the policies and the set of VMs that may be impacted by the instantiated event.
- Finally, when the actual event arrives, the proactive verification module matches it against the watchlist and the incremental verification module merges its effect to the actual state of the virtual infrastructure.

3.3 Proactive Verification

In this section, we present the details of the proactive verification module with the corresponding running examples.

Prediction. Our prediction algorithm/method works in two major steps: building dependency model and predicting future events.

- To build a probabilistic dependency model, we first collect historical data (e.g., event logs) from the cloud platforms, and then process those logs to identify corresponding events; finally, we calculate conditional probabilities to construct the dependency model using a Bayesian network.
- To predict future events, we leverage the already built dependency model, which captures each event transition and its corresponding probabilities. To that end, we intercept each run-time event from the cloud management platform, then measure the conditional probability of each possible future transition from the the current event, and finally, identify the most probable future event(s) for our proactive verification.

Pre-computation. Once our prediction step is completed, we assume those predicted events as the candidates for our pre-computation step. Any event exceeding the predefined threshold becomes a candidate for pre-computation. A candidate event gets instantiated based on all possible parameters, as demonstrated through an example based on OpenStack [17] in Table 1. To illustrate this, Figure 7 shows how VMGuard works on our running example; the upper part of the figure is the timeline for incoming events and the actions taken by VMGuard, and the lower part of the figure shows the dependency model (left), the network state (center), and the watchlist for the event E_N (right).

Example 1. From the sequence of events in Figure 7, event E_{K-1} creates a VM (*ID: 2134*). According to the dependency model, the next event could be *delete VM* or *attach port*. Assume the threshold is 0.5 in all our examples. The pre-computation module selects *attach port* as the next event to be evaluated in advance. As shown in Table 1, to instantiate this event, both *VM_ID* and *subnet_ID* are the required parameters. The event generator generates three possible events based on the current network state: *"Post create 2134, SN_{A1}"*, *"Post create 2134, SN_{A2}"*, and *"Post create 2134, SN_{A3}"*.

Proactive Verification. Two possible scenarios may happen for proactive verification, i.e., the actual event may either occur before, or after the pre-computation process finishes.

TABLE 1: Examples of reachability-related events, their parameters to be instantiated, and the number of possible instantiations

Event	Parameters	# of Predicted Parameters
Attach Interface	Router_ID and subnet_ID*	#Routers × #Subnets
Attach Port	VM_ID and subnet_ID	#Subnets
Attach Public IP	VM_ID and unallocated public IP	#VMs × #unallocated public IP
Delete Router	Router_ID	#Routers
Detach Interface	Router_interfaces_ID	#Router interfaces
Detach Port	VM_ port_ID	#Port
Detach Public IP	Allocated public IP	#Allocated public IPs
Detach Security Group Rule	Security_group_ID	#Security group rules

* Exception: A router should not have any overlapping subnets

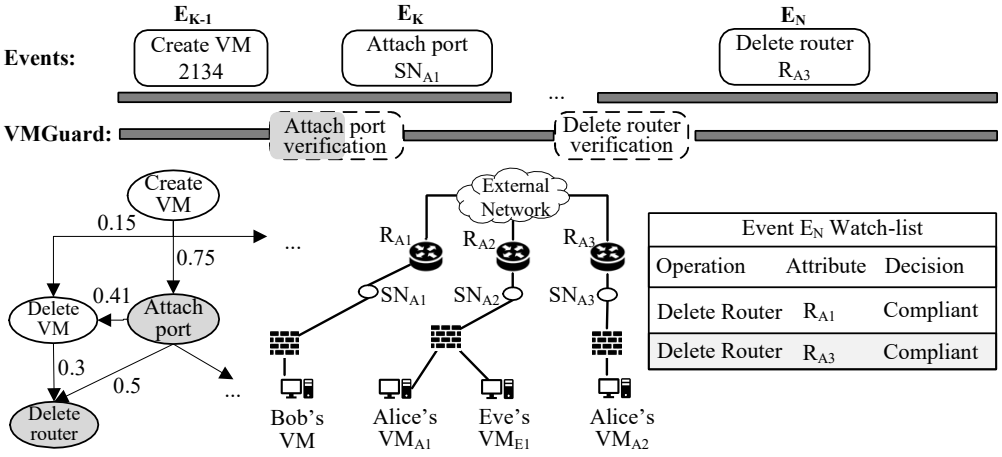


Fig. 7: Applying VMGuard on the running example

In the first case, VMGuard will switch to the intercept-check mode, which only verifies the intercepted event, e.g., *attach port verification* in Figure 7 (some other situations, e.g., a mistakenly predicted event, will also trigger the intercept-check mode). In the second case, VMGuard will trigger proactive verification, which searches for the event in the watchlist, e.g., *delete router verification*. In either case, the watchlist is flushed, as it is no longer valid with the state of the cloud. The verification process will be detailed in Section 3.4, and for now it can be regarded as a black-box.

Example 2. The next event for *attach port* is *"E_N: delete router"*. After instantiating the event with the three available routers, R_{A1}, R_{A2}, and R_{A3}, VMGuard triggers the incremental verification. The results of the verification are added to a watchlist, shown in Figure 7 (lower right). Assume the network isolation policy says *"VM_{E1} allows all ingress traffic from the external network"*, which means Eve's VM under R_{A2} must be reachable to the external network. Then, *"Delete router R_{A2}"* is a non-compliant event and will be removed from the watchlist, whereas *"Delete router R_{A3}"* is compliant and will be kept in the watchlist and allowed later when it occurs.

3.4 Incremental Verification

As we discussed in Section 3.1, *emulating* the effect of multiple predicted events requires creating copies of the state of the virtual infrastructure. For this purpose, we propose the **Fork** and **Merge** procedures in the *VMGuard* incremental verification module.

Fork. During the fork procedure, each instantiated event will be associated with an independent copy of the current state of the virtual infrastructure. To maintain the scalability of this solution, we will leverage the pruning module (detailed in Section 3.5) to limit the scope of verification and we will further evaluate the overhead of the fork procedure through experiments in Section 5. Once the instantiated event is applied to a copy of the state and the pruning module has been applied, we will employ the resultant state to verify the reachability against the given isolation policy, and then delete that copy of the state as soon as the verification completes. In this way, we do not incur the additional storage overhead for maintaining multiple copies of the state, and only the compliant results will be stored in the watchlist for proactive verification. The actual state of the virtual infrastructure remains intact until the merge procedure is triggered.

Merge. The merge procedure will be triggered when a compliant event is actually received. It will update the actual state of the virtual infrastructure based on the effect of this event.

Algorithm 1: INCREMENTAL VERIFICATION

```
   Input: Event, RadixTries, PrunedList,
          ReachabilityRelatedEvents
   Output: result∈{Compliant, Non-Compliant}
1  Copy RadixTries to RadixTriesFork
2  if event ∈ ReachabilityRelatedEvents then
3      Update RadixTriesFork with event
4      for VMdst ∈ PrunedList do
5          for VMsrc ∈ PrunedList do
6              Triepub ←
                 getBTrie(VMdst.publicIP.CIDR,VMsrc.subnet_id)
7              Triepriv ← getBtrie(VMdst.private.CIDR, rounter_id)
8              routable ← Route-Lookup(Triepub,Triepriv)
9              if routable is true then
10                 if VerifyPolicy(VMsrc,VMdst) is true then
11                     result ← Compliant
12                 else
13                     result ← Non-Compliant
14                     break
15 return result
   Input: event_response, Radixtries
16 if event_response is success then
17     Update RadixTries with event
```

In Algorithm 1, Lines 10-15 present the fork procedure. It takes an event, the current radix tries [26] (a data structure that flattens IP address ranges and arranges them in a manner to efficiently search routes), the *prunedList* (this list will be generated in Algorithm 2 in Section 3.5) and *ReachabilityRelatedEvents* (See Table 1 as an example). Lines 1-3 in the algorithm generate a copy of the state (*RadixTriesFork*) for a reachability event and apply the event to the

state. Lines 4-5 take the pair of VMs from the prunedlist, then Lines 6-8 evaluate the reachability between two VM pairs. The function *getBtrie* reads radix Tries to collect both (*Triepub*) public and (*Triepriv* private routes. The function *Route-Lookup* finds routes based on binary tries. Function *VerifyPolicy* checks whether the reachability between VM_{src} and VM_{dst} is allowed. Then Lines 9-14 generate compliance results by comparing the policy to the reachability results of TenantGuard. If a non-compliant result is found, this instantiated event is marked as *non-compliant*. Lines 16-17 show the merge procedure only update the state when the *event_response* is *success*.

Example 3. In Figure 7, E_{K-1} triggers proactive verification for event E_K. The fork procedure emulates the impacts of three events on three independent copies of the state. However, since the actual event *"attach port* SN_{A1}*"* occurs before the pre-computation phase completes, the verification against other copies gets flushed immediately; the verification happens only on the copy of state with the actual event. State-based verification tools, e.g., TenantGuard, will identify the new reachability in the copied state between the newly created VM (*ID: 2134*) and Bob's VM. Assume the policy is *"*VM_{Bob}*, *, Deny"* (i.e., denying all ingress traffic to this VM), which means this actual event is non-compliant and therefore, will be blocked by *VMGuard*.

3.5 Pruning

To improve the scalability of our incremental verification, we present the pruning module in this section. The main purpose of this module is to reduce the number of VMs for incremental verification. Only the compliant results corresponding to the pruned list of VMs are stored in a watchlist to wait for the actual event. However, the pruning step is optional and should be applied by a user if s/he is only concerned with vulnerabilities that may cause changes only in a specific part of the cloud.

In Algorithm 2 (pruning procedure), Lines 1-3 list the VMs that have at least one policy associated with the input event's tenant ID. The list *VMsUnderTenantPolicy* consists of the VMs covered by the policies specified by tenants and the list *VMsUnderEventScope* are VMs which are affected by the current event among those covered by the policies. If the list is not empty, Lines 4-11 also list the VMs that might be affected by the input event by comparing the attributes of the events, e.g., the type and the parameters of the event. In the end, a pruned list is generated with the intersection of the two VM lists in Line 11 and this result will be returned to the incremental verification module.

Example 4. We illustrate how pruning works for Example 3. We first check the policy*"*VM_{Bob}*,*, Deny"* to identify Bob's VM as the only VM under the policy. Since the *attach port* event may affect reachability under the same subnet, which applies to Bob's VM, the pruning module intersects the two lists and generates a pruned list with only Bob's VM. As a result, instead of verifying all the eight pairs of reachability, the verification will only need to be performed between Bob's VM and the newly created VM.

Algorithm 2: PRUNING

```
   Input: Event, Policies, RadixTries
   Output: PrunedList
1  for each Policy ∈ Policies do
2      if Policy.TenantID = Event.TenantID then
3          Add Policy.VMs to VMsUnderTenantPolicy
4  if VMsUnderTenantPolicy is not empty then
5      if event is a Routing event then
6          for each RouterInterface ∈
             RadixTries.RouterInterfaces do
7              for each port ∈ RounterInterface do
8                  Add get(VM) to VMsUnderEventScope
9      else
10         Add get(VM) to VMsUnderEventScope
11     PrunedList ←
         VMsUnderTenantPolicy∩VMsUnderEventScope
12 return PrunedList
```

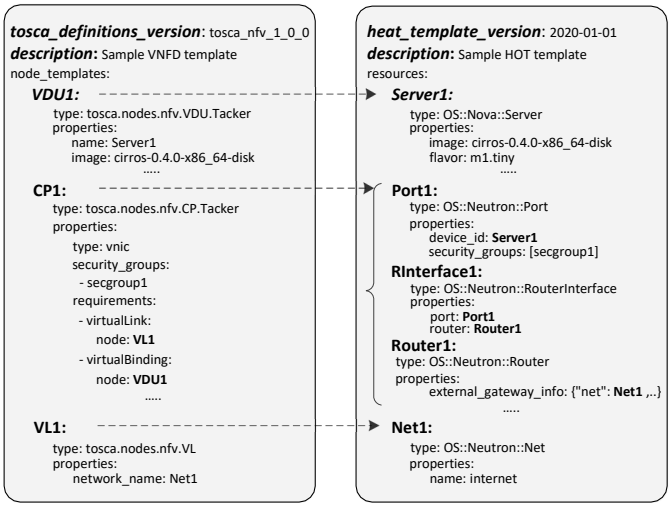


Fig. 8: Example of TOSCA [24] translated into HOT [25] i.e., interpretable by VMGuard.

3.6 NFV Adaptation

The design of VMGuard is meant to handle verification of network isolation both in Cloud and NFV environments. Therefore, the design presented earlier in this section does not require any change to adapt it to the NFV environment. The main effort is to identify the NFV-specific policies related to network isolation from the NSD written in TOSCA. In the following, we describe this in more details.

NFV Policy Recommender. VMGuard utilizes the network described in HOT to identify all isolation policies for the tenant by analyzing all-pair reachability to other tenants. The identified policies are presented to the tenant in order to select the applicable ones. Those policies are then enforced either proactively or in the intercept-check mode as discussed in Section 3.1.

Figure 8 illustrates an example of a mapping between TOSCA network service requirements and HOT definition of the virtual resources. The latter is composed of virtual resources such as VMs, ports, routers, routing interfaces,

etc., representing the tenants network. Algorithm 3 uses HOT templates to generate the policies. Therein, in Line 1, virtual resources requested by heat and expressed in YAML format are extracted. Lines 2-4 show queries to collect the requested network resources (virtual machines, ports, security group rules, routing interfaces, and router) and add them to a list *NetworkResource*. A state fork in Line 5 and all-pair reachability in-scope of tenant is derived in Line 6 using function *IntraTenantAllPair*. Reachability is identified in Lines 8-11, based on which policies are produced with corresponding actions. Line 12 returns the policies applicable to network isolation within the tenant.

Algorithm 3: POLICY RECOMMENDER

```
   Input: Heat Orchestration Template
   Output: Policies
1  Resources = ParseYAML(HOT)
2  for each Resource ∈ Resources do
3      if Resource.Type ==
        (Server/Subnet/Port/Router/Interface/Security
        Group) then
4          NetworkResource=Resource;
5  Graph = Fork(NetworkResource);
6  Reachability = IntraTenantAllPair(Graph);
7  for each Result ∈ Reachability do
8      if Result==Reachable then
9          Policies=policies(Result,Allow);
10     else
11         Policies=policies(Result,Deny);
12 return Policies
```

Example 5. A TOSCA extract in Figure 8 (left), comprises a section of network service configurations, i.e., a VNF. The VNF component (i.e., VDU) directly maps to a cloud resource VM i.e., VM_{Bob} in our motivating example. Some TOSCA components are abstraction of cloud resources. For instance, a Connection Point (CP) abstracts $Port_{A3}$, R_{A1}, etc. Similarly, all components from TOSCA map to HOT, represented in Figure 8 (right).

4 IMPLEMENTATION

VMGuard is implemented as a module deployed on top of OpenStack [17], a widely used open-source cloud management platform, and Tacker [27], the official OpenStack NFV MANO project. The latter builds a VNFM and an NFVO on top of Opentack as a VIM. We also installed prerequisites such as Heat [27], Mistral [27] and Barbican [27]. OpenStack Heat deploys workflows that are created using Mistral and Barbican, which facilitates secure communications.

Figure 9 illustrates the high-level architecture of VMGuard. There are mainly three phases. First, the initialization phase is for pre-processing the given policies and dependency models, which is conducted only once. Second, the run-time phase is for event-driven verification and is conducted with each successive event. Third, the audit phase is to ensure the correctness and performed periodically. Each phase is detailed as follows.

Initialization Phase. Before initialization, the dependency model is generated for each tenant with the logs collected from two OpenStack services, Nova (compute) and Neutron (network). The raw logs are processed to feed into the Bayesian network tool, SMILE & GeNIe [28] in order to generate the dependency model. The configuration information from the Nova and Neutron databases are processed to generate the radix tries. The heat templates are fetched and network isolation policies are identified by the policy processor (as discussed in Section 3.6). Using dependency model, VMGuard identifies the tenants infrastructure and proposes related network isolation policies. The provided policies are converted into the intra-tenant format, as discussed in Section 6, and an initial verification is conducted to check the compliance of the current cloud state.

TABLE 2: Statistics of the datasets

Dataset	Tenants	VMs	Routers	Subnets	Policies per tenant
DS1	50	4,362	300	525	100
DS2	100	10,168	600	1,288	200
DS3	150	14,414	800	1,828	300
DS4	200	20,207	1,000	2,580	400
DS5	250	25,246	1,200	3,210	500

Run-Time Phase. An audit middleware [29] is used to intercept each Neutron and Nova events. We leverage the TenantGuard [14] implementation, written in Java, and extend it with our pruning module. We also implement the deep object copy [30], and multi-threading [31] for pre-computation (on multiple copies of the state in parallel). The intercept-check verification threads are executed with a higher priority because they require immediate processing, whereas the pre-computation thread executes with a lower priority. The threads are initialized with the tenant IDs to assist the thread management. At any moment, either the pre-computation thread or the intercept-check thread is executed for a tenant.

Audit Phase. An audit phase is performed periodically as a thread with a low priority and ID as *audit* to distinguish it from run-time threads. The audit phase performs sync-check, state check and log learning, discussed in Section 6. A sync-check is performed by graph comparison [32], using JGraphT [33], a Java graph library, to verify the graphical representation of the state. The state check is performed using a special verification fork that takes corresponding states and all the policies as input. The log learner collects Nova and Neutron logs from Ceilometer [34], the telemetry service in OpenStack, and converts them to the input format for GeNIe to update the tenant's dependency model.

5 EXPERIMENTS

In this section, we first describe experimental settings in Section 5.1 and then present experimental results with both real and synthetic data in Section 5.2.

5.1 Experimental Settings

Our test cloud is an OpenStack release Mitaka with one controller node and 80 compute nodes. Each node runs Ubuntu 16.04 server on an Intel i7 dual-core CPU with 2GB memory. The Neutron network driver is L2 OpenVSwitch with the L3 agent plugins, which is a popular networking

deployment. To perform the experiments we run VMGuard over the controller node.

We use the cloud schema presented in the recent OpenStack survey [35] as a basis for our simulation. Our datasets are derived based on data collected from a real virtual infrastructure deployed in-house by one of the largest telecommunication vendor. Table 2 describes the statistics about our datasets that quantifies the number of tenants, virtual resources and policies per tenant. We simulate an environment with maximum 250 tenants and 25,246 VMs, which OpenStack survey [35] state as largest size cloud. We conduct the experiment on five different datasets varying the number of tenants from 50 to 250, policies per tenant from 100 to 500, subnets from 500 to 3,200, routers from 300 to 1,200, while keeping the number of VMs fixed to 100 per tenant. We believe these datasets represent a wide-range of real-life medium to large size cloud setups. Every experiment is performed at least 100 iterations.

Table 3 lists the events that we used in the experiments, which can be categorized into three levels, namely, VM-level, subnet-level and router-level. Each event can affect the verification results differently as shown in the third column. Simply, the higher the network hierarchy level of the component affected by the event is, the larger is the number of affected resources. This set of events is used in evaluating VMGuard.

TABLE 3: Events used in the experiments in three levels, namely, VM-level, Subnet-level, and Router-level, with the resources, which verification is affected

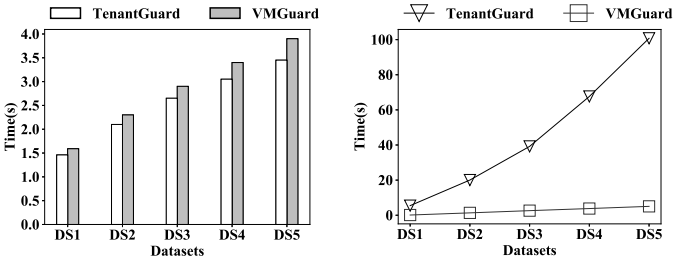
Levels	Events	Affected Resources
VM-level	Create VM	None
	Delete VM	only the deleted VM
	Attach Security Group	only the VM with this security group
	Delete Security Group	only the VM with this security group
	Attach Public IP	None
	Detach public IP	only the VM with this IP
Subnet-level	Attach Port	all VMs under the subnet
	Delete Port	all VMs under the subnet
	Create Network	None
	Delete Network	all VMs under the subnet
Router-level	Create Router	None
	Delete Router	all VMs reachable through this router
	Create Interface	None
	Detach Interface	all VMs reachable through this router

5.2 Performance of VMGuard

We evaluate VMGuard in this section from multiple phases with different enabled modules. In this section, we refer $VMGuard_0$ as the implementation of event-driven incremental state-based verification tool (disabling both pruning and proactive modules of VMGuard), $VMGuard_1$ as the version of VMGuard with only proactive module disabled, and $VMGuard_2$ as the complete version of VMGuard (incremental + pruning + proactive).

5.2.1 Initialization Phase

We first compare the performance of our work with the state-of-art solution, TenantGuard [14] for the initialization phase. All the experiment results are gathered from a single machine. Parallelization can still be applied to further improve the performance, which is considered as future work. The first set of experiments compare the time consumption for the initialization phase including data collection and initial verification.



(a) Data Collection (b) Initialization

Fig. 10: The time consumption for VMGuard and TenantGuard [14] in (a) data collection, (b) the initialization phase

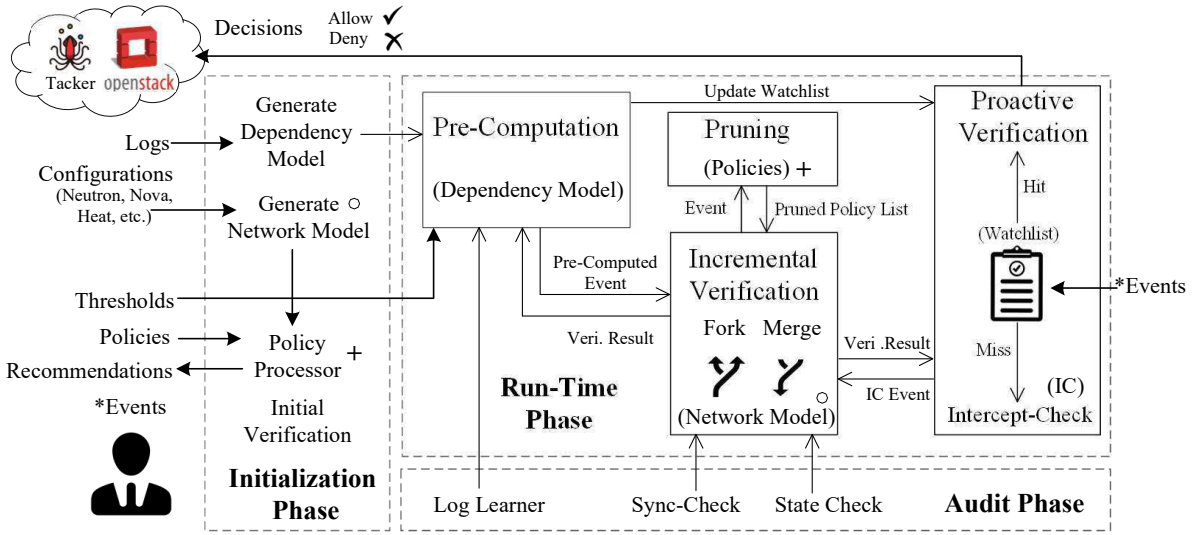


Fig. 9: The high-level architecture of VMGuard

Results and Implications for Initialization. In general, the one-time initialization phase takes longer than other phases. Due to the extra policy processing module, compare to TenantGuard, VMGuard requires a slightly longer time to finish data collection as shown in Figure 10(a). However, VMGuard performs much better in initialization verification. We observe that TenantGuard's initialization time increases exponentially with the size of the cloud, whereas VMGuard shows negligible increase. This is mainly because the scope of reachbility generation for TenantGuard is directly correlated with the size of the cloud, while the scope of the verification for VMGuard only relies on the tenant specified policies, which will remain within a reasonable size comparing with the full size of the cloud.

5.2.2 Incremental Verification and Pruning

TenantGuard is originally designed to work on the static snapshot of the virtual infrastructure [14]. Making TenantGuard incremental and suitable for an event-driven application faces many implementation challenges (some of these are demonstrated in Section 3.1). In implementing VMGuard, we have tackled those challenges and, by disabling both the pruning and proactive modules of VMGuard, we basically obtain an incremental version of TenantGuard, i.e., $VMGuard_0$. In the second set of the experiments, we compare $VMGuard_0$ (incremental) and $VMGuard_1$ (incremental+pruning) with events selected from different hierarchical levels of the virtual infrastructure. Figures 11(a), (b) and (c) correspond to VM-level, subnet-level, and router-level events, respectively.

Results and Implications for Incremental Verification and Pruning. Both $VMGuard_0$ and $VMGuard_1$ perform verification within a promising time range (0.4s to verify a high complexity event, router level event that impacts more VMs to be verified, in the largest dataset) for different types of events. We can observe that $VMGuard_1$ takes significant less time than $VMGuard_0$ in all cases, which clearly demonstrates the benefit of the pruning module. When comparing among the three figures, we can see the time consumption for $VMGuard_0$ and $VMGuard_1$ is similar in both Figures 11(a) and (b). The main reason behind this observation is that, although *attach security group rule* and *attach port* are the events at two different levels, the scope of these two events is similar (i.e., the VMs that directly correspond to the security group rule or the port). Other updating/deletion/addition events at these two levels are expected to share a similar trend. Also, as shown in Figure 11(c), the *delete router* event requires significantly longer verification time, because the verification depends on the number of subnets under the router and the number of VMs under each subnet; the scope of the verification is therefore larger than with the previous two events (even the *add router* event would generate less verification overhead than the *delete router* event).

5.2.3 Proactive Verification

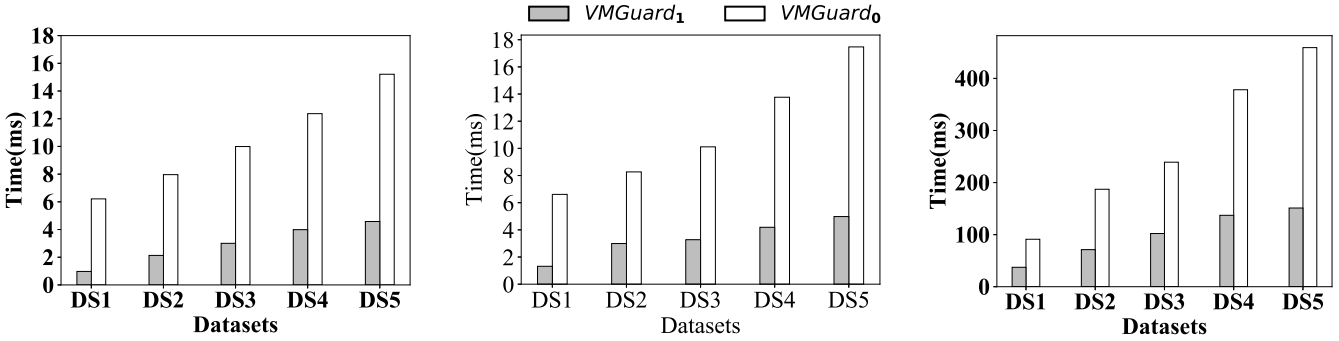
The third set of the experiments evaluates the efficiency of the proactive module, namely, $VMGurad_2$ (the complete version of VMGuard with all modules enabled). Since whether the proactive verification is triggered depends on the given threshold, $VMGuard_1$ can be considered as a special case of $VMGuard_2$ by setting the threshold as 1 (means no event can trigger the proactive module). In this experiment, we use real data collected from a real world community cloud hosted at one of the largest telecommunications vendors to obtain the dependency model and extract the sequences of events.

Results and Implications for Efficiency of Proactive Verification. Figure 12(a) shows the verification time comparison between the $VMGuard_1$ (intercept-check) and the $VMGuard_2$ (proactive). The latter one requires significantly less verification time, which demonstrates the benefit of the proactive module. The events used in this experiment span three levels, and our experiment results show that the higher hierarchical events, e.g., router-level events, require less processing time in $VMGuard_2$ than the lower hierarchical events, e.g., VM-level events. This mainly because higher hierarchical events naturally contain less number of predicted events than the lower hierarchical events.

The upper figure of Figure 12(b) shows the run-time memory consumption of $VMGuard_0$ as well as the results of $VMGuard_1$. The high memory consumption for $VMGuard_0$ is mainly due to its need to maintain a large number of pair-wise reachability. After the initialization phase, the memory consumption stays nearly plateau because each successive event will only affects a limited amount of reachability. Comparing $VMGuard_1$ to $VMGuard_0$, pruning shows a positive correlation with memory and CPU consumption as less pair-wise reachability needed to be calculated; thus, $VMGuard_1$ shows that it requires only 30% of the CPU and memory of $VMGuard_0$. Since $VMGuard_1$ has less event to calculate, different than $VMGuard_0$ it shows a drop in the CPU consumption during idle time.

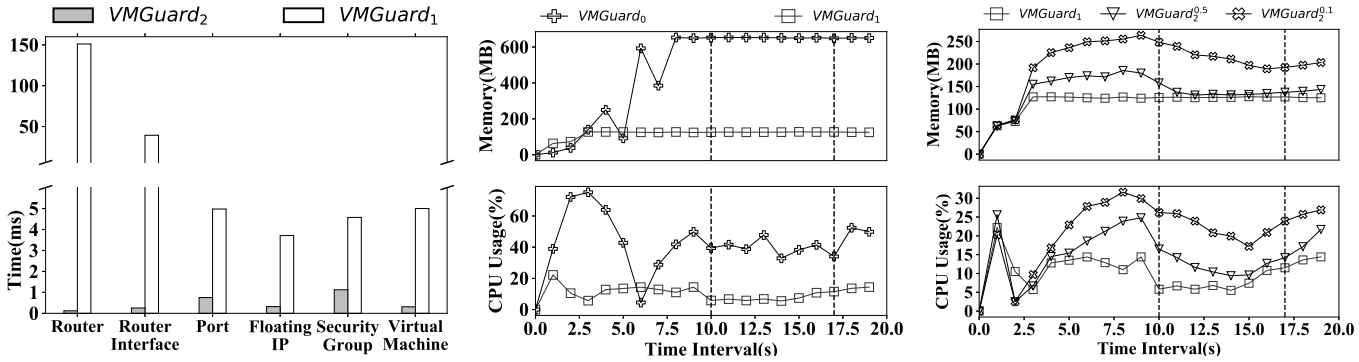
The proactive module in VMGuard shorten the process time for the incoming event by consuming more CPU and memory to precalculate multiple predicted events. In this set of the experiment, we evaluate the cost of proactive verification among various thresholds.

Results and Implications for Cost of Proactive Verification. The upper figure of Figure 12(c) shows the run-time memory consumption of $VMGuard_2$ under two different thresholds, as well as the results of $VMGuard_1$, which is the special case of $VMGuard_2$ with the threshold equals to 1. Comparing $VMGuard_2$ (under both threshold values) to $VMGuard_1$, the number of triggered pre-computations show a positive correlation with the memory consumption. During the idle time, the time intervals between 10s to 17.5s, ${VMGuard_2}^{0.5}$ finishes the pre-computation and shares the same memory consumption as $VMGuard_1$; however, due to the larger number of pre-computation candidate events, ${VMGuard_2}^{0.1}$ still shows a higher memory consumption during idle time to finish the ongoing pre-computation tasks. The CPU consumption shares almost the same trend as the memory consumption. It is also worth to notice, even though comparing to $VMGuard_1$, both $VMGuard_2$d require more memory and CPU to precalculate predicted event, they still require less resources than $VMGuard_0$ as shown in Figure 12(b).



(a) Attach Security Group (b) Attach Port (c) Delete Router

Fig. 11: The performance of $VMGuard_0$ (incremental) and $VMGuard_1$ (incremental+pruning) for (a) VM-level, (b) subnet-level, and (c) router-level events



(a) Time consumption (b) $VMGuard_0$ vs. $VMGuard_1$ (c) $VMGuard_1$ vs. $VMGuard_2^{0.5}$ vs $VMGuard_2^{0.1}$

Fig. 12: The performance for (a) $VMGuard_2$ and $VMGuard_1$ vs. Time, (b) $VMGuard_1$ and $VMGuard_0$ vs. Memory (up) and CPU (bottom), and (c) $VMGuard_2$ with different thresholds (1, 0.5, 0.1) vs. Memory (up) and CPU (bottom)

5.2.4 Scalability

Finally, we conduct this experiment to test the scalability of VMGuard. As shown in Figure 12, the proactive module significantly reduces the performance time of VMGuard. For evaluation purposes, we disable this module in the scalability tests. Note that this configuration represents the worst case scenario, i.e., $VMGuard_1$.

Results and Implications for Scalability. Overall, we achieve promising results while scaling up the network with different structure and configurations with different types of events. In this experiment, we observe that the router event (e.g., *delete router*) and router interface event (e.g., *detach interface*) require more time during the verification. This is mainly because each event in router or router interface level is associated with a larger number of verification candidates than a event happen in a lower hierarchical level. In VMGuard, the intercept check module verifies all the VMs in the pruned list, which means the larger pruned list, the longer verification time; when the pruned list stays the same, the verification time would be constant as well. As we discussed in Section 3.5, the pruned list is directly associated with two lists: the number of VMs under the policy and the number of VMs under the impact of the event. In Figure 13(a), the number of VMs under the policy stays the same, however, the impacted number of VMs under router and router interface event increases with the number of VMs per subnet. Therefore, we can observe the increase of time consumption for both types of events. Since other events mainly contain VM level events, the number of impacted VMs is the same as the number of VMs under the policies. Thus, the verification time stays the same while the number of VMs per subnet increases. Different than Figure 13(a), only the impact of router event increases with the number of subnet per router in Figure 13(b); we observe the increase of router event and a constant trend for the other two events. In Figure 13(c), the number of VMs under the policy increases since the number of policy per tenant increases. In other words, the number of pruned list increases in all level of events; we observe the increase of time consumption for all the events.

6 Discussion

Effect of a Wrong Prediction. VMGuard pre-computes verification for a predicted event. If there is a wrong prediction resulting from the inaccuracy of our dependency model, the pre-computed results are not useful and hence, VMGuard works as an intercept-check solution in such circumstances. To continuously improve the accuracy of our dependency model, we periodically update this model from cloud logs

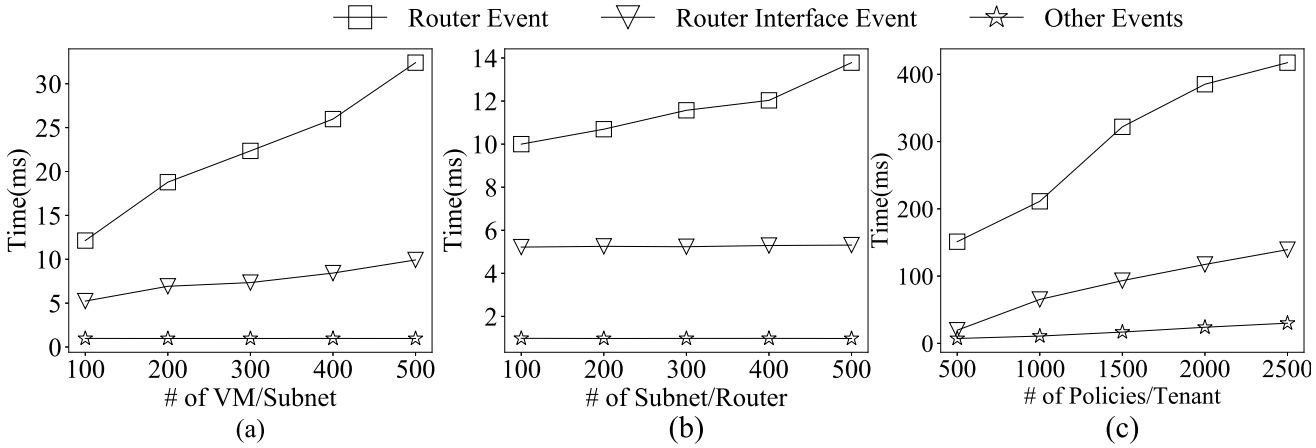


Fig. 13: Performance comparison by varying the # of (a) VMs per subnet, (b) subnet per router, and (c) policies per tenant

TABLE 4: Comparing existing solutions with VMGuard

Proposal	Methods	Feature								Scope	
		Retroactive	I-C	Proactive	Incr.	Paral.	Real-Time	Pruning	M-Thread	Vir. Net.	D. Plane
Weatherman [15]	Graph-theoretic		✓	✓	✓					✓	-
Congress [20]	Datalog			✓						-	-
PVSC [36]	Custom algorithms			✓	✓		✓			-	-
LeaPS [16]	Custom + Bayesian			✓	✓		✓			-	-
NoD [13]	Datalog	✓			✓			✓		✓	✓
Plotkin et al. [37]	SMT Solver	✓			✓					✓	✓
Madi et al. [38]	CSP Solver	✓								✓	✓
Cloud Radar [39]	Graph-theoretic	✓								✓	-
TenantGuard [14]	Custom algorithms	✓			✓	✓				✓	✓
VMGuard	Custom algorithms		✓	✓	✓	✓	✓	✓	✓	✓	✓

in the audit phase. In addition, to be more accurate in the prediction, we incorporate structural dependencies imposed by the cloud platform (e.g., a subnet cannot be deleted before detaching all its ports).

Choice of Threshold Values. In VMGuard, the amount of pre-computation effort is controlled through a threshold value. A threshold is provided by the cloud provider based on experiences, using which cloud provider can control the prediction rate based on his/her cloud workload.

Correctness of VMGuard Inputs. The regular operation of VMGuard relies on the correctness of the extraction of event parameters from the cloud management API, and identification of reachability effecting events. Therefore, to detect any flaws in those steps and further improve them, VMGuard introduces an audit phase, which periodically performs a full (not incremental) state-based verification.

Supported Policies by VMGuard. VMGuard is designed to verify policies related to virtual network isolation. A list of security policies (defined by the cloud provider and/or its tenants) is an input to the VMGuard system. These policies can vary in nature, such as inter-tenant (i.e., cross-tenant reachability), and intra-tenant. As an example, VMGuard supports the list of policies proposed in NoD [13]. For the ease of pruning and verification, VMGuard converts an inter-tenant policy into two intra-tenant policies where destination is the external network.

Cross-Platform Portability. Our work can potentially be extended to other cloud platforms, such as Amazon EC2 [40], Google GCP [41], Microsoft Azure [42] and VMware vCD [43]. Similar to TenantGuard [14], the virtual infrastructure model can be portable over different cloud platforms. Also, The similarities between management APIs and cloud logs in different cloud platforms show the potentiality of adapting our solution to those platforms [44], [45], [46], [47], [48].

Attacks Exploiting Unknown Vulnerabilities. As a state-based verification tool, VMGuard can identify violation of a predefined security policy, e.g., reachability between two specific VMs (e.g., VM_1 and VM_2) is not allowed. The violation could be caused by a misconfiguration, or exploiting a known or unknown vulnerability. VMGuard will not identify the root cause (i.e., which vulnerability is exploited) for this violation, but any operation that would cause such a violation could be blocked by VMGuard. Therefore, in this sense, VMGuard is able to monitor and tackle known and unknown vulnerabilities that may lead to a potential violation against user defined policies.

7 Related Work

Table 4 summarizes the comparison between existing works and VMGuard. The first and second columns enlist existing works and their verification methods, respectively. The next eight columns compare these works according to different features, i.e., retroactive, intercept-check (I-C), proactive, incremental (Incr.), parallel workload distribution (Paral.), pruning-based verification, and multi-threading to manage multiple requests (M-Thread). The last two columns compare the scope of network isolation works, i.e., virtual networks (Vir. Net.), and data plane (D. Plane) in such networks. The main benefit of VMGuard over those works is

that VMGuard provides a real-time response while verifying virtual network isolation in the data plane. To that end, VMGuard's unique combination of features is: proactive, incremental and pruning.

Network Isolation Verification. There exist several works (e.g., [13], [14], [37], [49]) for virtual network isolation verification. Among them, NoD [13], Plotkin et al. [37] and TenantGuard [14] adopt a retroactive approach, which detects an isolation breach after the fact. Specifically, NoD [13], is a logic-based verification engine that checks reachability policies using Datalog. Plotkin et al. [37] leverage the regularities existing in data centers to lessen the verification overhead using bi-simulation and modal logic. However, both of these works cause hours to days delay in verifying reachability. Whereas, TenantGuard [14] achieves verification time of 18 minutes for the same dataset by performing a hierarchical verification approach. However, for policy verification, TenantGuard relies on Congress [20], which causes a significant delay (as discussed in Section 2). Unlike these works, VMGuard achieves a practical response time (e.g., in few milliseconds), as reported in Section 5 by adopting a proactive approach. There exist some other works (e.g., [49], [50], [51], [52], [53], [54]) for SDN-based or traditional networks. Among them, NetPlumber [49] leverages verifying hypotheses before deploying, but it's only applicable to SDN-based networks. In contrast, *VMGuard* verifies hypothesis in the virtual network environment.

Proactive Verification. There exist few proactive verification solutions (e.g., [15], [16], [20], [36]) for clouds. Weatherman [15] performs proactive verification on the virtual infrastructure based on the future change plan. Similarly, Congress [20] performs proactive verification over the proposed hypothetical configuration for the cloud. Both of those works rely on manual identification of future plan, and otherwise, cause a significant delay as an intercept-and-check solution. Whereas, VMGuard adopts an automated proactive approach (based on dependency model), and achieves response time in few milliseconds. Similar to VMGuard, PVSC [36] and LeaPS [16] achieve response time in milliseconds. However, those works rely on signatures and cannot detect many isolation breaches (as demonstrated in Section 3). To that end, VMGuard adopts a state-based approach and hence, overcomes this limitation.

8 Conclusion

In this paper, we addressed the major issues (e.g., inefficiency and inaccuracy) in the existing virtual network isolation verification approaches and proposed VMGuard, which is a state-based proactive approach to efficiently verify network isolation policies in a large scale virtual infrastructure. To achieve better efficiency, VMGuard proactively conducted the verification of future events. On the other hand, to ensure the effectiveness, VMGuard simulated all possible impacts on the current state and verified all those simulated states. We also showed how VMGuard can be seamlessly used to verify network isolation between network services deployed by multiple tenants in NFV. Furthermore, we integrated VMGuard with OpenStack for Cloud and NFV deployments, and evaluated its performance and efficiency through extensive experiments using both real and synthetic data. As future work, we will investigate the feasibility of integrating a formal policy specification language with our solution to enhance its policy support.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported partially by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair (IRC) in SDN/NFV Security.

References

[1] M. Ali, S. U. Khan, and A. V. Vasilakos, "Security in cloud computing: Opportunities and challenges," *Information Sciences*, vol. 305, pp. 357–383, 2015.

[2] Cloud Security Alliance, "Security guidance for critical areas of focus in cloud computing v 4.0," available at: https://cloudsecurityalliance.org/working-groups/security-guidance.

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] ETSI, "Network Functions Virtualisation," available at: https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[6] R. Kumar and R. Goyal, "On cloud security requirements, threats, vulnerabilities and countermeasures: A survey," *Computer Science Review*, vol. 33, pp. 1–48, 2019.

[7] O. Ali, J. Soar, and J. Yong, "Challenges and issues that influence cloud computing adoption in local government councils," in *IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD'17)*, 2017, pp. 426–432.

[8] V. D. Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "A survey of network isolation solutions for multi-tenant data centers," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 4, pp. 2787–2821, 2016.

[9] Amazon Web Services, "Amazon web services: Overview of security processes," 2017, available at: https://d1.awsstatic.com/whitepapers/aws-security-whitepaper.pdf.

[10] ISO Std IEC, "Iso 27002:2005. information technology-security techniques," 2005, available at: http://www.iso27001security.com/html/27002.html.

[11] ——, "Iso 27017. information technology- security techniques," 2013, available at: http://www.iso27001security.com/html/27017.html.

[12] Cloud Security Alliance, "Cloud control matrix ccm v3.0.1," 2014, available at: https://cloudsecurityalliance.org/research/ccm/.

[13] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, 2015, pp. 499–512.

[14] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation," in *Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[15] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim, "Proactive security analysis of changes in virtualized infrastructures," in *Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[16] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Leaps: Learning-based proactive security auditing for clouds," in *European Symposium on Research in Computer Security (ESORICSs'17)*, 2015, pp. 265–285.

[17] Openstack, "Openstack : Cloud operating system," 2019, available at: https://www.openstack.org/.

[18] OpenStack, "Ossa-2014-008: Routers can be cross plugged by other tenants," available at: https://security.openstack.org/ossa/OSSA-2014-008.html.

[19] ——, "Nova network security group changes are not applied to running instances," 2015, available at: https://security.openstack.org/ossa/OSSA-2015-021.html.

[20] OpenStack, "Openstack congress," 2019, available at: https://wiki.openstack.org/wiki/Congress.
[21] ETSI, "Network FunctionsVirtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Ve-Vnfm Reference Point," available at: https://www.etsi.org/deliver/etsi_gs/NFV-SOL/001_099/002/02.03.01_60/gs_NFV-SOL002v020301p.pdf.
[22] OpenStack, "OpenStack Documentations," available at: https://docs.openstack.org/.
[23] OASIS, "Open standards, Open source," available at: https://www.oasis-open.org/.
[24] ——, "TOSCA Simple Profile for Network Functions Virtualization (NFV)," available at: http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html.
[25] OpenStack, "Heat : OpenStack Orchestration," available at: https://docs.openstack.org/heat/latest/template_guide/hot_guide.html.
[26] J. Corbet, "Trees i:radix trees," 2018, available at: https://lwn.net/Articles/175432.
[27] Oracle, "Tacker - OpenStack NFV Orchestration," available at: https://wiki.openstack.org/wiki/Tacker.
[28] BayesFusion, "Genie and smile," 2019, available at: https://www.bayesfusion.com.
[29] Cloud auditing data federation, "Pycadf: A python-based cadf library," 2019, available at: http://docs.openstack.org/developer/keystonemiddleware/audit.html.
[30] Wikipedia, "Object copying," 2019, available at: https://en.wikipedia.org/wiki/Object_copying\#Deep_copy.
[31] Oracle, "Java: Processes and threads," 2019, available at: https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html.
[32] R. J. W., "Definitions and examples," in *Introduction to Graph Theory, Second Edition*, 1979.
[33] JGraphT, "A java library of graph theory data structures and algorithms," 2019, available at: https://jgrapht.org/.
[34] OpenStack, "Openstack ceilometer project," 2019, available at: https://docs.openstack.org/ceilometer/latest/.
[35] Openstack, "Openstack user survey," 2017, available at: https://www.openstack.org/assets/survey/April2017SurveyReport.pdf.
[36] S. Majumdar, Y. Jarraya, T. Madi, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Proactive verification of security compliance for clouds through pre-computation: Application to openstack," in *European Symposium on Research in Computer Security (ESORICS'16)*, 2015, pp. 47–66.
[37] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Principles of Programming Languages (POPL'16)*, 2016.
[38] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang, "Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack," in *Conference on Data and Application Security and Privacy (CODASPY'16)*, 2015, pp. 195–206.
[39] S. Bleikertz, C. Vogel, and T. Groß, "Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures," in *Annual Computer Security Applications Conference (ACSAC'14)*, 2015, pp. 26–35.
[40] Amazon Web Serives EC2, "Aws ec2," 2019, available at: https://aws.amazon.com/ec2/.
[41] Google, "Google cloud platofrm," 2019, available at: https://cloud.google.com/.
[42] Microsoft, "Microsoft azure," 2019, available at: https://azure.microsoft.com/en-us/.
[43] VMware, "VMware Cloud," 2019, available at: https://cloud.vmware.com/.
[44] OpenStack, "Openstack networking api v2.0," 2018, available at: https://developer.openstack.org/api-ref/network/v2.
[45] ——, "Openstack networking api v2.0," 2018, available at: https://docs.aws.amazon.com/networkmanager/latest/APIReference/API_Operations.html.
[46] ——, "Openstack networking api v2.0," 2018, available at: https://cloud.google.com/compute/docs/reference/rest/v1.
[47] ——, "Openstack networking api v2.0," 2018, available at: https://docs.microsoft.com/en-us/rest/api/virtual-network/.
[48] ——, "Openstack networking api v2.0," 2018, available at: https://code.vmware.com/apis/722.
[49] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013, pp. 99–111.
[50] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013, pp. 15–27.
[51] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012, pp. 113–126.
[52] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *ACM SIGCOMM (SIGCOMM'16)*, 2016, pp. 300–313.
[53] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. D. Millstein, "A general approach to network configuration analysis," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, 2015, pp. 469–483.
[54] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, 2014, pp. 87–99.

Gagandeep Singh Chawla Gagandeep Singh Chawla is currently working as Security Architect with SAP Security, Montreal, Canada. He received his master's degree in information system security from Concordia University, Montreal, Canada. His research interests include security of public/private clouds, containers, container orchestration platforms and Software Defined Networks (SDN).

Mengyuan Zhang Mengyuan Zhang is an Experienced Researcher at Ericsson Research, Montreal, QC, Canada. She received her Ph.D. in Information and Systems Engineering from Concordia University in Montreal. Her research interests include security metrics, attack surface, cloud computing security, and applied machine learning in security. She has published several research papers and book chapters on the aforementioned topics in peer-reviewed international journals and conferences such as TIFS, TDSC, CCS, ESORICS.

Suryadipta Majumdar Suryadipta Majumdar is currently an Assistant Professor in Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Canada. Previously, Suryadipta was an Assistant Professor in the Information Security and Digital Forensics department at University at Albany – SUNY, USA. He received his Ph.D. on cloud security auditing from Concordia University. His research mainly focuses on cloud security, Software Defined Network (SDN) security and Internet of Things (IoT) security.

Yosr Jarraya Yosr Jarraya is a Master Researcher at Ericsson since 2016 focusing on security and privacy in cloud, SDN and NFV. She received a Ph.D. in electrical and computer engineering from Concordia University Montreal, Canada, in 2010. She has several patents granted or pending. She co-authored two books and over 40 research papers in peer-reviewed international journals and conferences such as TOPS, TIFS, TDSC, JCS, NDSS, and ESORICS.

Makan Pourzandi Makan Pourzandi received the M.Sc. degree in parallel computing from École Normale Supérieure de Lyon, France, and the Ph.D. degree in computer science from the University of Lyon I, France. He is currently a Researcher with Ericsson, Canada. He has over 15 years of experience in security for telecom systems, cloud computing, distributed systems security, and software security. He is the inventor of over 28 patents granted or pending. He has published over 50 research papers in peerreviewed scientific journals and conferences.

Lingyu Wang Lingyu Wang is a professor in the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Quebec, Canada. He holds the NSERC/Ericsson Industrial Research Chair (IRC) in SDN/NFV Security. He received his Ph.D. degree in Information Technology in 2006 from George Mason University. His research interests include SDN/NFV security, cloud computing security, network security metrics, software security, and privacy. He has co-authored seven books, two patents, and over 100 refereed conference and journal articles including many published at top journals/conferences, such as TOPS, TIFS, TDSC, TMC, JCS, S&P, CCS, NDSS, ESORICS, PETS, ICDT, etc. He is serving as an associate editor for IEEE Transactions on Dependable and Secure Computing (TDSC) and Annals of Telecommunications (ANTE) and an assistant editor for Computers & Security, and he has served as the program (co)-chair of seven international conferences and the technical program committee member of over 150 international conferences.

Mourad Debbabi Mourad Debbabi is a Full Professor at the Concordia Institute for Information Systems Engineering and Interim Dean at the Gina Cody School of Engineering and Computer Science. He holds the NSERC/Hydro-Quebec Thales Senior Industrial Research Chair in Smart Grid Security and the Concordia Research Chair Tier I in Information Systems Security. He is also the President of the National Cyber Forensics and Training Alliance (NCFTA) Canada. He is a member of CATAAlliance's Cybercrime Advisory Council. He serves/served on the boards of Canadian Police College, PROMPT Québec and Calcul Québec. He is the founder and one of the leaders of the Security Research Centre at Concordia University. Dr. Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published 5 books and more than 300 peer-reviewed research articles in international journals and conferences on cyber security, cyber forensics, smart grid, privacy, cryptographic protocols, threat intelligence generation, malware analysis, reverse engineering, specification and verification of safety-critical systems, programming languages and type theory. He supervised to successful completion 32 Ph.D. students, 76 Master students and 14 Postdoctoral Fellows. He served as a Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Canada; Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France.