NFVGuard: Verifying the Security of Multilevel Network Functions Virtualization (NFV) Stack

Alaa Oqaily*, Sudershan L T*,Yosr Jarraya†, Suryadipta Majumdar*, Mengyuan Zhang†,
Makan Pourzandi†, Lingyu Wang*, Mourad Debbabi*
**Concordia Institute for Information Systems Engineering, Concordia University*, Montreal, QC, Canada
Email:{a_oqaily, s_akshma, smajumdar, wang, debbabi}@encs.concordia.ca
†Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
Email:{yosr.jarraya, mengyuan.zhang, makan.pourzandi}@ericsson.com

***Abstract*—Network Functions Virtualization (NFV) enables agile and cost-effective deployment of multi-tenant network services on top of a cloud infrastructure. However, the multi-tenant and multilevel nature of NFV may lead to novel security challenges, such as stealthy attacks exploiting potential inconsistencies between different levels of the NFV stacks. Consequently, the security compliance of a multilevel NFV stack cannot be sufficiently established using existing solutions, which typically focus on one level. Moreover, the naive approach of separately verifying every level could be expensive or even infeasible. In this paper, we propose, NFVGuard, the first multilevel approach to the formal security verification of NFV stacks. Our key idea is to conduct the security verification at only one level, and then assure that verification result for other levels by verifying the consistency between adjacent levels. We integrate NFVGuard with OpenStack/Tacker, a popular platform for the NFV deployment, and experimentally evaluate its effectiveness.**

***Index Terms*—NFV, Multilevel Verification, Tacker, OpenStack, Formal Verification, Topology Consistency**

I. Introduction

The popularity of NFV is on the rise (e.g., 60% of network service providers will be adopting NFV by 2021 [25]). By virtualizing proprietary physical devices in the network architecture, NFV allows operators to scale their network capabilities on demand and with a lower cost. On the other hand, an NFV stack is typically a complex system that involves multiple levels of virtualization, and the managerial components could operate at each level autonomously, namely, the "split-brain" design [3]. Such added complexity could become a double-edged sword that leads to novel security threats, such as security breaches at lower levels of an NFV stack that are invisible to end users [16]. The potential inconsistencies between different levels mean that, to ensure the security compliance of an NFV stack, security properties (e.g., network isolation) must be valid across all levels.

To that end, most existing works (e.g., [7], [8], [21], [29], [30], [37], [38]) are insufficient since they typically focus on one particular level of the NFV stack, such as service function chaining (SFC). Moreover, a naive solution to separately verify every level of the NFV stack would require each security property to be interpreted and re-defined at all levels, which could be expensive or even infeasible in practice. We illustrate this limitation and our key idea through a motivating example.

Motivating Example. The left side of Figure 1 shows a simplified view of the NFV stacks of two tenants, *Bob* and *Eve* (as indicated by the two dashed line boxes), which involve four levels (as depicted by the shaded areas). Knowing that a malicious tenant, such as *Eve*, could potentially inject a malicious virtual machine `VM5` into *Bob*'s network (e.g., to secretly inspect his traffic) directly at L3, without causing any detectable changes at upper levels [1], the provider is concerned with the following question: *"Are Bob's and Eve's virtual networks properly isolated at all levels?"*

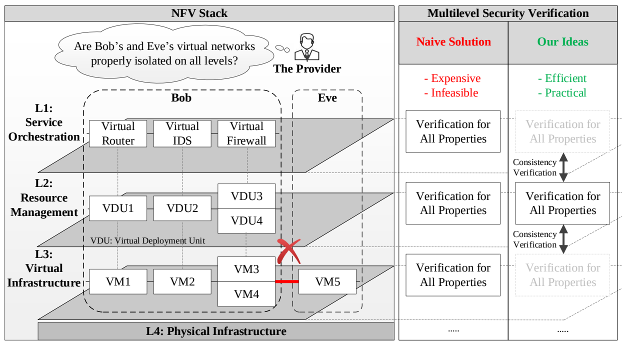

NFV Stack
Are Bob's and Eve's virtual networks properly isolated on all levels?
The Provider
Bob
Eve
L1: Service Orchestration
Virtual Router
Virtual IDS
Virtual Firewall
L2: Resource Management
VDU1
VDU2
VDU3
VDU4
VDU: Virtual Deployment Unit
L3: Virtual Infrastructure
VM1
VM2
VM3
VM4
VM5
L4: Physical Infrastructure
Multilevel Security Verification
Naive Solution
Our Ideas
- Expensive
- Infeasible
- Efficient
- Practical
Verification for All Properties
Consistency Verification

Fig. 1: A motivating example to highlight the challenge of multilevel security verification in NFV and provide a hint of our idea

As shown on the right side of the figure, a naive solution is to simply verify all four levels. However, this solution can become expensive with the growing number and complexity of security properties. Furthermore, it may not always be feasible to interpret and re-define a security property across two non-adjacent levels (e.g., re-defining a L1 property at L4). To that end, our key idea is to verify every property at the level where it is specified (e.g., L2 in this case), and then implicitly extend such verification results to other levels by verifying the consistency between adjacent levels.

More specifically, we propose a multilevel security verification approach, namely, *NFVGuard*, to assure security properties for all levels of an NFV stack. First, we identify NFV-related security properties and consistency properties by

[1]By exploiting real-world vulnerabilities, e.g., CVE-2015-3456 [22], CVE-2015-7835 [22], or CVE-2018-10853 [22] in a specific way [16].

studying relevant security standards and NFV specifications (e.g., IETF-RFC7498 [12] and ETSI [3]). Second, we develop our verification approach by collecting relevant configuration data from different components, correlating data at each level, and aggregating data between different levels. Third, we utilize formal methods to verify the compliance of the system against security and consistency properties and provide audit evidences. Finally, to demonstrate the applicability of our approach, we implement our solution based on an OpenStack/Tacker [24] testbed, and evaluate its efficiency through experiments using both real and synthetic data. In summary, our main contributions are as follows.

- As per our knowledge, we are the first to propose a novel multilevel security verification approach that can ensure a security property is valid throughout the NFV stack without explicitly verifying it at all levels.
- We design our solution for formal verification by collecting, correlating and aggregating audit data from the different NFV stack levels, and by leveraging a Constraint Satisfaction Problem (CSP) solver, Sugar [28].
- We implement and integrate our solution into OpenStack/Tacker [24], which is a popular platform to deploy NFV [23], and our experimental results demonstrate its efficiency and practicality.

The remainder of the paper is organized as follows. Section II provides the background, the threat model and the challenges. Section III presents our methodology. Section IV details the implementation and reports on experimental results. Section V reviews the related work. Finally, Section VI concludes the paper.

II. Preliminaries

This section first provides a background on NFV, and then defines our threat model and identifies the challenges.

A. Background on NFV

NFV is a network architecture concept that virtualizes various network functions, such as routers, firewalls, load balancers, and intrusion detection systems (IDS) [5]. The left side of Figure 2 illustrates the ETSI NFV reference architecture [5], which depicts two abstraction levels, namely, the *Virtual Network Function (VNF)* level, which provides a high-level representation of network functions, and the *NFV Infrastructure (NFVI)* level, which represents the underlying cloud infrastructure. The right side of Figure 2 illustrates the multilevel NFV deployment model [16], which complements the ETSI architecture with deployment details found in multiple open source platforms. Specifically, the deployment model depicts the NFV stack at four abstraction levels, i.e., *Service Orchestration (L1)* (which supports the specification, on-boarding, and lifecycle management of network services), *Resource Management (L2)* (which supports the instantiation of network services and the management of compute, storage, and network resources), *Virtual Infrastructure (L3)* (which hosts the virtual resources needed to support upper levels, and optionally the SDN controller (SDN-C)), and *Physical Infrastructure (L4)* (which includes all the physical resources). Table I lists the NFV-specific terms that will be used later in the paper.

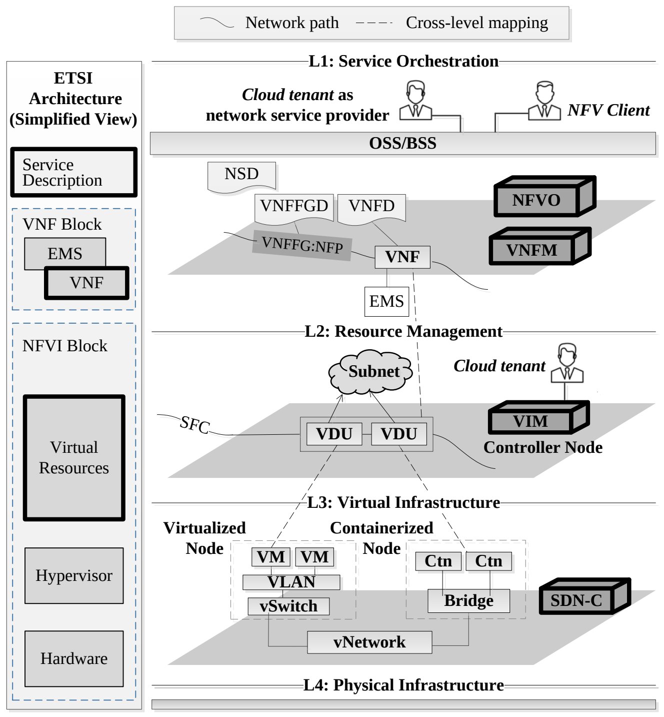


Fig. 2: The multilevel NFV model [16]

Acronym	Full Name		Acronym	Full Name
CP	Connection Point		SDN-C	SDN Controller
VIM	Virtual Infrastructure Manager		VDU	Virtual Deployment Unit
OVS	Open vSwitch		NFP	Network Function Path
NFVO	NFV Orchestrator		VNF	Virtual Network Function
NS	Network Service		VNFD	VNF Descriptor
NSD	Network Service Descriptor		VNFFG	VNF Forwarding Graph
PPG	Port Pair Group		VNFFGD	VNFFG Descriptor
SFC	Service Function Chain		VNFM	VNF Manager

TABLE I: Summary of the NFV-specific terms used in the paper

B. Threat Model and Challenges

Our in-scope threats include both external attackers who exploit existing vulnerabilities in the NFV stack, and insiders such as cloud users and tenant administrators who cause security breaches either by mistakes or with malicious intentions. Similar to most security verification solutions, we assume the integrity of the audit input data collected through logs and database records and we trust the NFV provider. Therefore, out-of-scope threats include attacks that do not cause any visible violation of the specified security properties, and attacks by those adversaries who can remove or tamper with logged events. Finally, although a security verification solution can detect a violation of security properties, it is not designed to attribute such a violation to underlying vulnerabilities (i.e., vulnerability analysis) or specific attacks (i.e., intrusion detection).

The security verification of an NFV stack exhibits several unique challenges. First, as explained in Section II-A, an NFV stack is a complex system with many inter-dependent entities located at different abstraction levels. Therefore, verifying its security requires a systematic understanding of the semantics of all the entities and their relationships inside the NFV stack. Second, to verify a given security property, we first need to identify all the involved data sources and what data to collect from each source, e.g., to verify the traffic isolation property of an SFC, data would need to be collected from the VDUs at L2, and from the VMs and vSwitches at L3. Third, since the data sources are typically scattered on multiple servers and across different levels, the collected data needs to be correlated and aggregated, e.g., following our previous example, data must be correlated among multiple OpenFlow tables from different servers, and the VDU, VM, and vSwitch data need to be aggregated across levels L2 and L3 based on their relationships. We will address those challenges in the coming section.

III. Methodology

This section first presents an overview of our multilevel verification approach, *NFVGuard*, and then elaborates on each of its main steps.

A. NFVGuard Overview

Figure 3 shows an overview of our approach. There are three major steps. First, we identify both NFV-related security and consistency properties from existing standards and literature (detailed in Section III-B). Second, we identify the data sources for verifying each security and consistency property, and design data processing techniques to correlate data at each level and aggregate data across different levels (detailed in Section III-C). Finally, we formulate the properties in the First Order Logic (FOL), instantiate them using the processed data, and verify them using a formal method tool (detailed in Section III-D).

B. Identifying Security and Consistency Properties for NFV

This step is to identify both security and consistency properties relevant to an NFV stack. For this purpose, we conduct extensive studies of both standards related to NFV (e.g., IETF-RFC7498 [12] and ETSI [3]), and standards related to various components of an NFV stack, such as cloud and SDN (e.g., ISO 27002 [10] and CCM [4]), since the security considerations of the latter will be inherited by the NFV stack. Table II shows examples of both security and consistency properties, their instantiation as sub-properties, descriptions of the sub-properties, and corresponding standards that require those properties for security compliance. Note that, although this list is not meant to be exhaustive, it can be easily extended to include other security and consistency properties, and even user-defined properties, as long as these can be formulated using FOL and verified through formal method, as shown in Section III-D.

C. Data Collection and Processing

In the following, we detail the data collection and processing steps.

Data Collection. To verify each security and consistency property, different data need to be collected from multiple sources across several levels in an NFV stack. The main data sources are logs and configuration files gathered from each level. For example, as shown in Figure 4, to verify the *VNFFG configuration consistency* between both L1/L2 and L2/L3, we need to collect the VNFFG specification from the Tacker database at L1 (depicted as part of a TOSCA file for simplicity), the data about ports, port pairs and port chains from the Nova and Neutron databases at L2 (depicted as a single table for simplicity), and the OpenFlow rules at L3 from multiple servers.

Data Correlation. Due to the distributed nature of virtual resources involved in a network service, the collected data usually need to be correlated within each level based on their relationships. For example, the bottom of Figure 4 shows that the VNFFG under verification is implemented at L3 as three VMs (VM_02, VM_04, and VM_05) hosted on two physical servers. The traffic steering information (which determines how traffic will flow through those VMs) needed for the verification is stored on both physical servers. Moreover, although not shown in the figure, such information is also scattered in many tables on each physical server (e.g., to check whether VM_04 is forwarding traffic to VM_05, we have to follow the forwarding rules stored in tables 0, 5, 10 and the Group table). Therefore, we need to correlate all those data in order to piece together sufficient information for the verification.

Data Aggregation. After correlating data at each level, we need to further aggregate the data across different levels of the NFV stack to verify consistency properties. For example, in Figure 4, by linking the VNFFG specification at L1 (left side of the figure) to the Nova and Neutron databases at L2 (upper right), and to the OpenFlow rules at L3 (bottom), we could finally verify that the VNFFG under verification indeed include one chain, consisting of three VNFs, implemented using three VMs, and hosted on two physical servers.

D. Formal Verification

To verify both security and consistency properties, we propose to formalize them and the audit data as a Constraint Satisfaction Problem (CSP), a time-proven technique for expressing many complex problems, and use Sugar [28], a well-established constraint solver, to check whether these properties are satisfied. Specifically, for each property, we encode the NFV system entities as the domains and instances of such entities as the CSP variables. For example, the entity *Tenant* is defined as a finite domain ranging over integer, such that `(domain TENANT 0 max_tenant)` is a declaration of tenant domain, where each value between `0` and `max_tenant` is for an instance. The variable `t` that corresponds to the *Tenant* instance (e.g., 18e552) will have a value within the domain `TENANT` (e.g., 10). The relationship between the system entities is encoded as an CSP relation with a support consisting of tuples of related instances, e.g.,

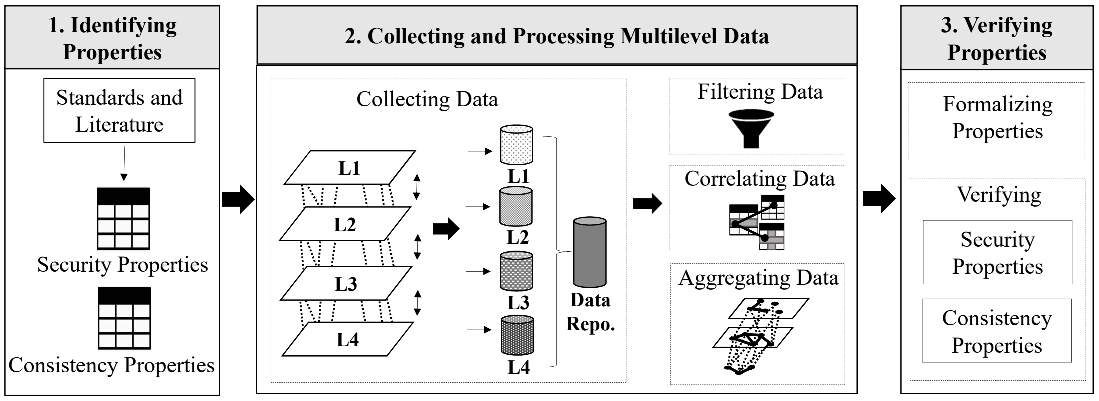


Fig. 3: An overview of the NFVGuard approach

Security Properties	Sub-Properties	Description	Standards
Topology isolation [18]	Mapping unicity VLANs-VXLANs	VLANs and VXLANs should be mapped one-to-one on a given server	ISO [11], NIST800 [27], CCM [4], ETSI [3], IETF-RFC7665, RFC7498 [12]
	Correct association Ports-Virtual Networks	VNFs should be attached to the virtual networks they are connected to through the right ports	
	Mappings unicity Ports-VLANs	Ports should be mapped to unique VLANs	
Physical resource isolation [18]	No VNFs co-residence	VNFs of a tenant should not be placed on the same compute node as VNFs of a non-trusted tenant	ISO [11], NIST800 [27], CCM [4], ETSI [3]
Virtual resource isolation [18]	No common ownership	Tenant-specific resources should belong to a unique tenant, unless permitted by a user-defined policy	CCM [4], ETSI [3], IETF-RFC7665, RFC-7498 [12]
Policy and state correctness [26]	-	A policy can be dynamically changing. The changed policy should be reconfigured in VNF node as soon as possible	ETSI [3], [6], IETF-RFC7665, RFC8459 [12]
Functionality of VNF and VNFFGs [7], [37]	-	Check if VNFs and the composition (i.e., service chaining) of these functions work as intended	ETSI [6], IETF-RFC-7665, RFC8459 [12]
SFC ordering and sequencing as defined by the specification [8]	-	SFCs should maintain the order of VNFs with the correct traffic forwarding behavior as defined by the specifications	ETSI [3], [6], IETF-RFC7665, RFC8459 [12]
Consistency Properties	**Sub-Properties**	**Description**	**Standards**
Topology consistency [18], [19]	VNFFG configuration consistency between L1/L2	Consistency between the size of VNFFGs, the sequences of VNFs and the classifiers at L1 and their parallel SFCs and classifiers definitions at L2	ISO [11], NIST800 [27], CCM [4], IETF-RFC-8459 [12], ETSI [3], [6]
	VNFFG configuration consistency between L2/L3	Consistency between the created SFCs and classifiers at L2 and their implementation at L3 i.e., the same number of created VMs with the correct order and traffic steering	

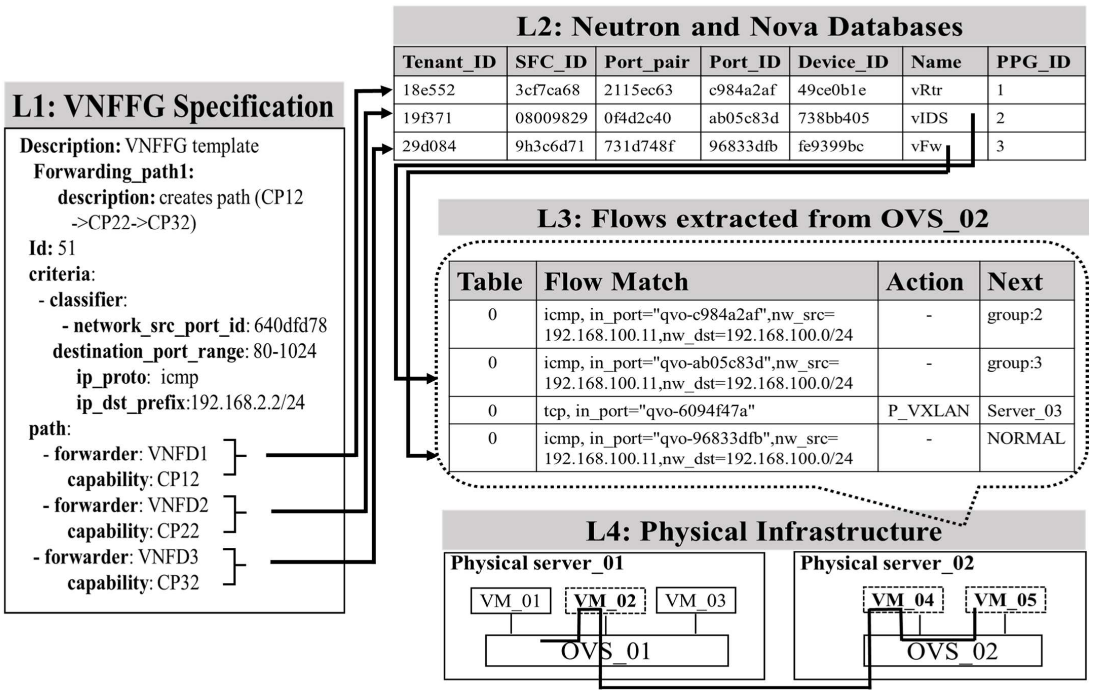
TABLE II: Examples of consistency and security properties in NFV

the relation between the tenant and the SFC that he creates is encoded as `(relation HasChain 2 (supports(t1 sfc1)(t2 sfc2)))`. Those CSP relations describe the current state of the system.

Each property is then expressed as predicates over the relations forming a CSP constraint. Those predicates correspond to the negation of the properties so that, when Sugar solves the constraints and finds no solution, those properties are reported to hold. Table III shows some example properties and their FOL expressions. For instance, for *VNFFG configuration consistency*, the FOL expression specifies that the VNFFG design at L1 is instantiated correctly as the corresponding SFC configurations at L2, and that the SFC configuration at L2 matches the traffic steering at L3. For the *virtual resource isolation*, the FOL expression specifies that the VDUs that composing a specific SFC at the management level are all owned by a unique tenant, i.e., the owner of the SFC service. For *Mapping unicity VLANs-VXLANs*, the FOL expression specifies that all network topologies are properly isolated, i.e., all VLANs associated with ports of the same virtual network on the same switch are mapped to a unique VXLAN across servers.

IV. Implementation and Experiments

In this section, we detail the implementation of NFVGuard and present the experimental results to evaluate the performance and overhead of our solution.



L1: VNFFG Specification

```
Description: VNFFG template
 Forwarding_path1:
    description: creates path (CP12
      ->CP22->CP32)
  Id: 51
  criteria:
   - classifier:
      - network_src_port_id: 640dfd78
    destination_port_range: 80-1024
      ip_proto: icmp
      ip_dst_prefix:192.168.2.2/24
  path:
   - forwarder: VNFD1
     capability: CP12
   - forwarder: VNFD2
     capability: CP22
  - forwarder: VNFD3
     capability: CP32
```

L2: Neutron and Nova Databases

Tenant_ID	SFC_ID	Port_pair	Port_ID	Device_ID	Name	PPG_ID
18e552	3cf7ca68	2115ec63	c984a2af	49ce0b1e	vRtr	1
19f371	08009829	0f4d2c40	ab05c83d	738bb405	vIDS	2
29d084	9h3c6d71	731d748f	96833dfb	fe9399bc	vFw	3

L3: Flows extracted from OVS_02

Table	Flow Match	Action	Next
0	icmp, in_port="qvo-c984a2af",nw_src= 192.168.100.11,nw_dst=192.168.100.0/24	-	group:2
0	icmp, in_port="qvo-ab05c83d",nw_src= 192.168.100.11,nw_dst=192.168.100.0/24	-	group:3
0	tcp, in_port="qvo-6094f47a"	P_VXLAN	Server_03
0	icmp, in_port="qvo-96833dfb",nw_src= 192.168.100.11,nw_dst=192.168.100.0/24	-	NORMAL

Fig. 4: An example of data collection, correlation, and aggregation

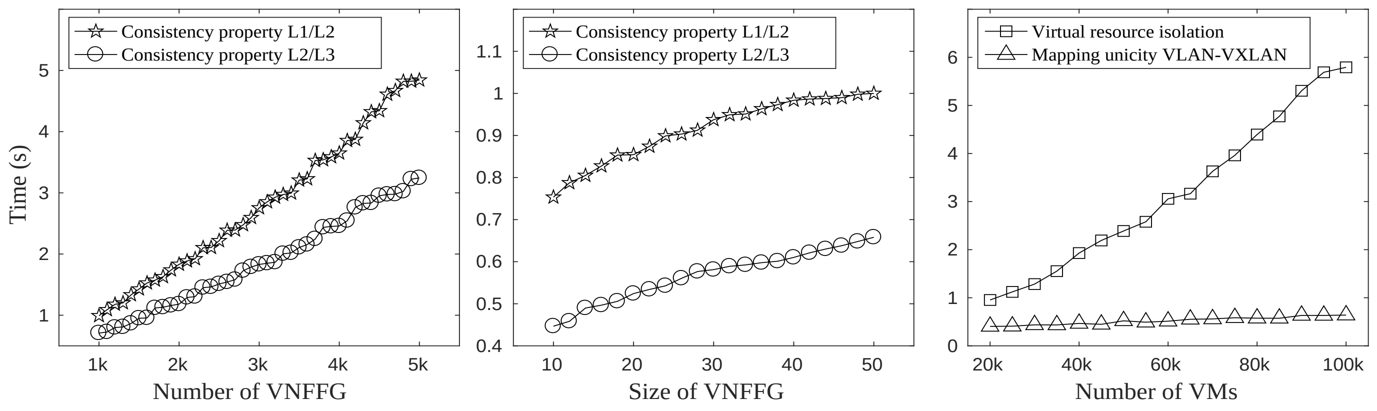
Security Properties	FOL Expression
Virtual resource isolation	$\forall t1 \in Tenant,\ \forall sfc \in SFC,\ \forall vdu1,\ vdu2 \in VDU:\ HasChain(t1,\ sfc) \wedge\ SFCHasVDUs(sfc,\ vdu1)\ \wedge\ SFCHasVDUs(sfc,\ vdu2) \implies HasVDU(t1,\ vdu1)\ \wedge\ HasVDU(t1,\ vdu2)$
Mapping unicity VLANs -VXLANs	$\forall vxlan1,\ vxlan2 \in VXLAN,\ \forall vlan \in VLAN,\ \forall sw \in OVS,\ \forall p \in PORT: IsAssignedVLAN(sw,\ p,\ vlan)\ \wedge\ IsMappedToVXLANOnOV(sw,\ vlan,\ vxlan1) \wedge\ IsMappedToVXLANOnOV(sw,\ vlan,\ vxlan2) \implies (not(vxlan1 = vxlan2))$
Consistency Properties	**FOL Expression**
VNFFG configuration consistency between L1 and L2	$\forall t1 \in Tenant,\ \forall fg \in VNFFG,\ \forall chain1 \in Chain,\ \forall path1 \in Path, \forall vnf1, vnf2 \in VNF, \forall cl \in Classifier, \forall s \in Src, \forall d \in Destination, \forall p \in Protocol: HasDefinedVNFFG(t1, fg)\ \wedge\ HasPath(fg,\ path1)\ \wedge HasClassifier(path1,\ cl,\ s,\ d, p)\ \wedge\ BelongsToPath(chain1,\ path1)\ \wedge HasVNFs(chain1, vnf1,\ vnf2) \Leftrightarrow\ HasDefinedSFC(t1,\ chain1)\ \wedge HasVDUs(chain1,\ vnf1,\ vnf2) \wedge HasClassifier(path1,\ cl,\ s,\ d,\ p) \wedge BelongsTo(vnf1,\ t1)\ \wedge\ BelongsTo(vnf2,\ t1)$
VNFFG configuration consistency between L2 and L3	$\forall t1 \in Tenant,\ \forall chain1 \in Chain,\ \forall path1 \in Path, \forall vdu1,\ vdu2 \in VDU, \forall cl \in Classifier,\ \forall s \in Src,\ \forall d \in Destination,\ \forall p \in Protocol: HasDefinedSFC(t1,\ chain1) \wedge SFCHasVDUs(chain1,\ vdu1,\ vdu2)\ \wedge HasClassifier(path1,\ cl,\ s,\ d,\ p) \Leftrightarrow CorrelatedFlows(s, d, p, vdu1, vdu2)$

TABLE III: Examples of security and consistency properties represented in First Order Logic (FOL).

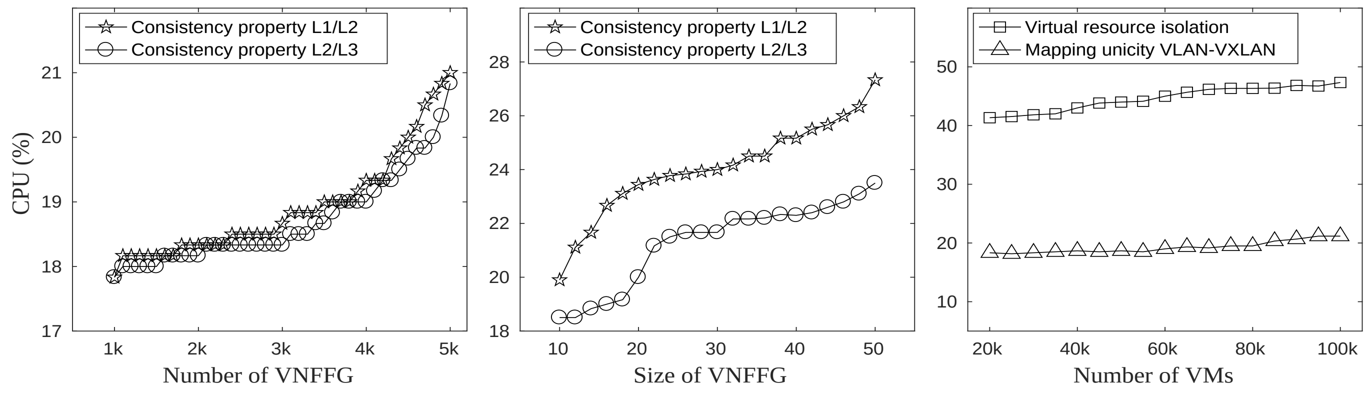
A. NFVGuard Implementation

The data collection component is implemented to collect data from different OpenStack services, such as Tacker, Nova [24], and Neutron [24], as well as from the Open vSwitch (OvS) instances running on every compute node. Specifically, we rely on the Tacker database to retrieve user-defined descriptors uploaded to the VNFM and NFVO modules of Tacker (such as VNFD and VNFFGD) as the basis for verifying most of the properties. We also rely on a collection of OpenStack databases, e.g., Neutron database for information about SFC networking (such as the sequence of service functions, the traffic steering in-between, and the traffic classifier) and Nova databases (table Instance) for information about the tenant, the VDU, and the hosting machine. Finally, we collect the OpenFlow tables and internal OvS databases from all the compute nodes, e.g., checking for inconsistencies between L2 and L3.

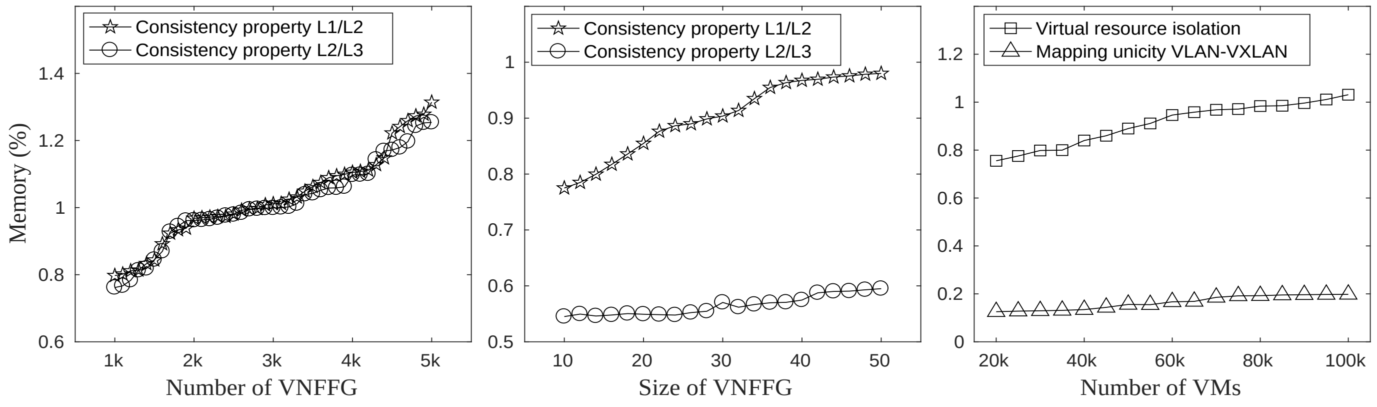
The data processing component is implemented in Python and Bash scripts as follows. First, for each property, our processing component identifies the involved relations, and the supports of the relations are either fetched directly from the collected data (such as the support of the relation *BelongsTo*) or recovered after data correlation. Second, our processing component formats each group of data as an n-tuple, i.e., (resource, tenant), (ovs, vlan, vxlan), etc. Finally, it uses the n-tuples to generate part of the Sugar source code, and appends it with the variable declarations, relationships, and predicates for each security property. A customized script is developed to generate the Sugar source code for the verification of each property. The formal verification component is implemented to feed the generated code into the Sugar SAT solver version 2.3.3 [28]. Sugar then produces the verification results to



(a) Time Consumption



(b) CPU Consumption



(c) Memory Consumption

Fig. 5: Verification performance for the topology consistency properties, virtual resource isolation, and mapping unicity VLANs-VXLANs by varying the number of VNFFGs (left), size of VNFFGs (middle), and VMs (right), respectively in (a) time, (b) CPU, and (c) memory consumption

either state the property holds or provide evidences when the property is breached.

B. Experiments

Experimental Settings. Our testbed is deployed on a SuperServer 6029P-WTR equipped with Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz and 128GB of RAM. To evaluate the performance of our solution, we generate 41 synthetic datasets of different sizes. We conduct experiments on those different datasets varying the number of VNFFGs from 1K to 5K, VNFs per VNFFG from 10 to 50 (with a fixed set of 100 VNFFGs), and VMs from 20K to 100K. Those datasets represent a reasonably large NFV setups according to the literature [9]. All data processing and experiments are conducted on the SuperServer with the verification tool, Sugar V2.3.3 [28], and each experiment is performed 1,000 times.

Our experiments show that the performance of NFVGuard largely depends on the properties to be verified. Therefore, we report results on both consistency and security properties that consume the most significant resources (time, CPU, and memory). More specifically, these include the VNFFG configuration consistency between VNFFGD at L1 and SFC at L2 (to ensure the correctness of VNFFG implementation), the VNFFG configuration consistency between SFC at L2 and OpenFlow rules at L3 (to ensure the correctness of SFC implementation), the virtual resource isolation (L2), and the mapping unicity VLANs-VXLANs (L3). Since the selected consistency properties depend on the VNFFG-related relations, we perform experiments by varying the number of VNFFGs and the size of VNFFG. On the other hand, since the security properties depend on the VM-related relations, we perform the experiments by varying the number of VMs.

Time Consumption. The objective of the first set of experiments is to evaluate the verification time required by the aforementioned properties. Figure 5(a) (left) and (middle) depict the verification time (in seconds) for the consistency properties, while Figure 5(a) (right) shows the verification time for the security properties. In general, the verification time is less than six seconds for all the properties in the largest dataset, and it increases almost linearly with the varied number of resources. The consistency property verification consumes more time for verifying between higher levels (L1 and L2, 1$\sim$5s) than for lower levels (L2 and L3, 1$\sim$3s). As to security properties, the verification of mapping unicity VLAN-VXLAN is more efficient (less than one second) and the time grows more slowly than it is for virtual resource isolation (six seconds for the largest dataset) as the latter has more complex predicates involving a higher number of relation instances.

CPU Consumption. The second set of experiments evaluates the CPU consumption for the aforementioned properties. In general, the CPU consumption (in %) shares a similar trend as the time consumption for most of the properties. We can observe the size of VNFFG affects the CPU consumption more than the number of VNFFGs. This is mostly because the CPU consumption relies on the variables to be verified, i.e., the number of VNFs. Note that the average VNFs per VNFFG is smaller in Figure 5(b) (left) than in Figure 5(b) (middle), and therefore, the CPU consumption increases more significantly in the latter (from $\sim$18% to $\sim$28%). The verification of the virtual resource isolation property consumes the most CPU ($\sim$40%), which again highlights the complexity of verifying this property.

Memory Consumption. The third set of experiments focuses on evaluating the memory consumption for the selected properties. Figure 5(c) depicts the memory usage (in %) for the verification of the aforementioned properties. In general, the memory consumption of all the properties is less than 1.3% of the 128GB of RAM on our server. Unlike the CPU consumption, the number of VNFFGs impacts the memory consumption more than the size of VNFFG does. This is mostly because the amount of relations stored in memory increases more significantly with the number of VNFFGs (0.8%$\sim$1.3%).

In summary, the experimental results show satisfactory efficiency and scalability of our approach, and demonstrate in general the feasibility of applying formal methods to NFV. Additionally, we note that the performance of NFVGuard can be further improved by distributing the verification tasks among multiple compute nodes as these properties can be verified independently from each other.

V. Related Work

Most existing solutions (e.g., [7], [8], [21], [29], [30], [37], [38]) in NFV focus on the verification of one particular level (mostly SFC). In particular, ChainGuard [8] and SFC-Checker [29] both verify the correct forwarding behavior of SFCs. ChainGuard checks the flow rules to verify the actual SFC traffic steering against the specifications, and SFC-Checker additionally analyzes the internal forwarding behavior of each network function. Moreover, vSFC [37] verifies a wide-range of SFC violations (e.g., packet injection attacks, flow dropping, and path non-compliance). Additionally, there exist several works (e.g., [7], [21], [30], [38]) on verifying the functionality and performance of SFCs. For instance, vNFO [7] verifies a wide-range of SFC functionalities (e.g., performance and accounting). Similarly, SLAVerifier [38] verifies the performance-related properties of SFC specified under SLA, such as delay, packet loss, jitter, and network availability. Marchetto et al. [21] propose an approach for generating optimal placement of SFCs based on given performance parameters (e.g., latency and CPU cycles) and formal verification of reachability policies within the required performance constraints. Wang et al. [30] propose a framework to automatically detect the dependencies and conflicts between network functions. Unlike all those works, the main focus of NFVGuard is to ensure the security of an NFV stack at all levels. Also, unlike NFVGuard, most of those works do not formally model the verification problem.

Also, there exist other works (e.g., [2], [13]–[15], [17], [20], [31]) that verify security properties in virtual networks, e.g., clouds and SDN. Among them, ISOTOP [18] and Xu et al. [32] cover the consistency between different cloud layers. Additionally, there are other solutions, e.g., NetPlumber [13], Veriflow [15], and NoD [17] that verify flow rules against various security and functionality properties in virtual networks. However, none of these works considers NFV, and extending them to NFV would require significant efforts due to the added complexity.

In the context of traditional networks, there are many works on verifying network functions (a.k.a. middleboxes) [1], [33]–[36]. Gravel [35] verifies the compliance of the middlebox implementation with its specification using satisfiability modulo theories (SMT). APKeep [36] incrementally updates the network model for real-time verification. Tiramisu [1] utilizes multi-layer graph model to encode the network and different custom algorithms for different categories of policies. Yousefi et al. [33] express different network functions in a compositional programming abstraction to capture network state changes. NetSMC [34] develops a customized symbolic model checking algorithms for verification based on the network model. However, all those methods are designed for traditional networks and do not take into consideration many intrinsic characteristics of virtual infrastructures such as a larger scale and significantly higher elasticity and dynamicity.

VI. Conclusion

We have presented NFVGuard, a novel multilevel approach to the formal security verification of NFV stacks. Specifically, we identified NFV-related security and consistency properties, designed our solution for data collection and processing techniques, and applied a formal method tool for verification. We implemented our solution and integrated it into our NFV testbed operating OpenStack/Tacker. We evaluated our approach through experiments using synthetic data. The results confirmed the efficiency and applicability of our approach.

Limitations and Future Work. First, the efficiency and scalability of NFVGuard are limited by the underlying formal method tool, and therefore, despite the satisfactory results demonstrated in our experiments, one can only afford to run NFVGuard periodically or on demand after a change. one future direction is to explore more efficient techniques using parallel, incremental, or proactive verification to further improve the efficiency and scalability. Second, in order to facilitate the verification we intend to build a system model that captures the system entities and their relationships in an NFV stack and study possibility to leverage this model to automate the specification of certain properties. Third, although the general approach of NFVGuard is platform-agnostic, the current implementation of data collection and processing is still limited to OpenStack/Tacker. Our future work will address this limitation through a more modular design with concrete methodology for extending to other open-source NFV platforms (e.g., OPNFV and OSM). Finally, our list of security and consistency properties is still limited in scope and not taking full advantage of the expressiveness of the underlying formal method. To address this limitation, we will continue building a more comprehensive repository of NFV-related properties to cover security from other angles (e.g., stateful VNFs).

References

[1] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast multilayer network verification. In *USENIX NSDI*, 2020.

[2] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim. Proactive security analysis of changes in virtualized infrastructures. In *ACSAC*, 2015.

[3] M. Bursell, A. Dutta, H. Lu, M. Odini, K. Roemer, K. Sood, M. Wong, and P. Wörndle. Network functions virtualisation (NFV), NFV security, security and trust guidance, v. 1.1. 1. In *Technical Report, GS NFV-SEC 003*. European Telecommunications Standards Institute, 2014.

[4] Cloud Security Alliance. Cloud control matrix CCM v3.0.1, 2014.

[5] ETSI. Network functions virtualisation architectural framework, 2013.

[6] ETSI. Network Functions Virtualisation (NFV); NFV Security; Problem Statement. *ETSI GS NFV-SEC*, 1, 2014.

[7] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar. Verifiable network function outsourcing: Requirements, challenges, and roadmap. In *HotMiddlebox*, 2013.

[8] M. Flittner, J. M. Scheuermann, and R. Bauer. ChainGuard: Controller-independent verification of service function chaining in cloud computing. In *NFV-SDN*, 2017.

[9] H. Hawilo, M. Jammal, and A. Shami. Exploring microservices as the architecture of choice for network function virtualization platforms. *IEEE Network*, 33(2):202–210, 2019.

[10] IEC, ISO Std. ISO 27002: 2005. *Information Technology-Security Techniques-Code of Practice for Information Security Management*, 2005.

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

[12] IETF, SFC. Internet Engineering Task, SFC Active WG Working Group Documents, 2020.

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

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

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

[16] S. Lakshmanan Thirunavukkarasu, M. Zhang, A. Oqaily, G. Singh Chawla, L. Wang, M. Pourzandi, and M. Debbabi. Modeling NFV deployment to identify the cross-level inconsistency vulnerabilities. IEEE CloudCom, 2019.

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

[18] T. Madi, Y. Jarraya, A. Alimohammadifar, S. Majumdar, Y. Wang, M. Pourzandi, L. Wang, and M. Debbabi. ISOTOP: Auditing virtual networks isolation across cloud layers in OpenStack. *ACM TOPS*, 22(1):1:1–1:35, 2018.

[19] 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 *ACM CODASPY*, 2016.

[20] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi. Security compliance auditing of identity and access management in the cloud: Application to OpenStack. In *IEEE CloudCom*, 2015.

[21] G. Marchetto, R. Sisto, J. Yusupov, and A. Ksentini. Virtual network embedding with formal reachability assurance. In *CNSM*, 2018.

[22] National Institute of Standards and Technology. CVE-2015-3456 Detail, 2015.

[23] OpenStack. Verizon launches industry-leading large OpenStack NFV deployment, 2016.

[24] OpenStack. OpenStack, 2020.

[25] Ovum. NFV Deployments Still on the Rise, 2020.

[26] M.-K. Shin, Y. Choi, H. H. Kwak, S. Pack, M. Kang, and J.-Y. Choi. Verification for NFV-enabled network services. In *ICTC*, 2015.

[27] SP, NIST. 800-53. *Recommended security controls for federal information systems*, pages 800–53, 2003.

[28] N. Tamura and M. Banbara. Sugar: A CSP to SAT translator based on order encoding. *Proceedings of the Second International CSP Solver Competition*, 2008.

[29] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang. SFC-Checker: Checking the correct forwarding behavior of service function chaining. In *NFV-SDN*, 2016.

[30] Y. Wang, Z. Li, G. Xie, and K. Salamatian. Enabling automatic composition and verification of service function chain. In *IWQoS*, 2017.

[31] 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 *NDSS*, 2017.

[32] Y. Xu, Y. Liu, R. Singh, and S. Tao. Identifying SDN state inconsistency in OpenStack. In *ACM SOSR*, 2015.

[33] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella. Liveness verification of stateful network functions. In *USENIX NSDI*, 2020.

[34] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar. NetSMC: A custom symbolic model checker for stateful network verification. In *USENIX NSDI*, 2020.

[35] K. Zhang, D. Zhuo, A. Akella, A. Krishnamurthy, and X. Wang. Automated verification of customizable middlebox properties with Gravel. In *USENIX NSDI*, 2020.

[36] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. APKeep: Realtime verification for real networks. In *USENIX NSDI*, 2020.

[37] X. Zhang, Q. Li, J. Wu, and J. Yang. Generic and agile service function chain verification on cloud. In *IWQoS*, 2017.

[38] Y. Zhang, W. Wu, S. Banerjee, J.-M. Kang, and M. A. Sanchez. SLA-verifier: Stateful and quantitative verification for service chaining. In *INFOCOM*, 2017.