Cross-Level Security Verification for Network Functions Virtualization (NFV)

Alaa Oqaily, Mohammad Ekramul Kabir, Lingyu Wang, Yosr Jarraya, Suryadipta Majumdar, Makan Pourzandi, Mourad Debbabi, Sudershan L T, Mengyuan Zhang

Abstract—Network Functions Virtualization (NFV) is a popular solution for providing multi-tenant network services on top of existing cloud infrastructures in an agile and cost-effective manner. However, as NFV employs multiple levels of virtualization, it also introduces novel security challenges, such as cloud-level security breaches that are invisible to NFV-level tenants. Towards verifying the security of NFV across all the levels (a.k.a. cross-level security verification), existing solutions are mostly insufficient, as each such solution typically only focuses on one specific level (e.g., cloud, SDN, or SFC), and verifying every level separately would be expensive or even infeasible. In this paper, we propose an efficient and practical system, *NFVGuard+*, for cross-level security verification for NFV. Particularly, the efficiency of *NFVGuard+* is achieved by first performing the costly security verification at one level, and then extrapolating the verification result to other levels through conducting relatively lightweight consistency checks. Additionally, the practicality of *NFVGuard+* is ensured by automating the essential steps (e.g., identifying security properties, collecting verification data, and conducting verification) based on a novel Entity-Relationship (ER) model of NFV stack, integrating the approach with OpenStack/Tacker (a popular choice for an NFV deployment), and finally evaluating its effectiveness using both synthetic and real data.

Index Terms—NFV, Cross-level Verification, Security Verification, Topology Consistency, Tacker, OpenStack.

1 Introduction

The adoption rate of NFV is increasing[1] due to the many benefits of virtualizing proprietary physical devices in the network architecture, such as the capability for operators to scale their network services on-demand, and the lower cost of using existing cloud infrastructure. However, to attain such benefits, NFV involves multiple levels of virtualization and operates the managerial components at each level autonomously [2]. As a ramification, this additional complexity opens the door to potential inconsistencies among different levels of the NFV stack, which can be exploited to conduct stealthy attacks, e.g., "invisible" (to end users) security breaches at lower levels of an NFV stack [3]. To tackle such threats, verifying the security across different levels of an NFV stack (a.k.a. cross-level security verification) becomes essential.

To that end, most existing works (e.g., [4]–[18]) are insufficient as they typically focus on one particular level of the NFV stack, such as service function chaining (SFC), instead of verifying the entire NFV stack. Additionally, utilizing those existing solutions to separately verify each level of NFV would be expensive, or even infeasible (as doing so would require translating given security properties to all NFV levels, which is not always possible). On the other hand, developing a new approach to cross-level verification for NFV involves the following major challenges: (i) how to determine the system entities and their relationships at multiple levels in NFV to locate the possible data sources for verification, (ii) how to instantiate the high-level security requirements (e.g., network isolation) into specific system-level security properties to enable automated verification in NFV, and (iii) how to conduct the cross-level verification in an efficient and accurate manner while handling the sheer size and multi-level of NFV. In the following, we further highlight those challenges using a motivating example.

Motivating Example. The left side of Figure 1 shows a simplified view of the NFV stack (Sec 2.1 provides more background on NFV stack) of two tenants, *Bob* and *Eve* (as indicated by the two dashed line boxes), which involve four levels (L1-L4 as indicated by the shaded planes) and their corresponding virtual and physical resources. We assume that, by exploiting real-world vulnerabilities (e.g., CVE-2024-1085 [19], CVE-2024-0193 [20], or CVE-2024-0646 [21]) in a specific way [3], a malicious tenant (*Eve*) could inject a malicious virtual machine, `Malicious VM`, into *Bob*'s network to secretly inspect his traffic at L3, without causing any detectable changes in the upper levels. Knowing about such potential threats, the provider is concerned with the following question: *"Are Bob's and Eve's virtual networks properly isolated at all levels?"*

The first column on the right side of the figure shows the existing challenges in cross-level security verification as follows. First, the mapping between the resources across different levels of the NFV stack (which might be useful for cross-level security verification) is unknown. Second, a naive solution which separately conducts security verification at each level of NFV through utilizing (multiple) existing works (e.g., [11], [13], [15], [16]) is expensive or even infeasible (e.g., its not always possible to re-define an L1 property at L4 in a meaningful way). To address those challenges, our two main ideas are illustrated in the next two columns of the figure. Specifically, our first idea is to identify the mapping between the resources in different levels of NFV and automatically identify the corresponding consistency properties (needed for the next idea). Our second idea is to only verify every property at the level where its specified

1. 92% of carriers have either deployed or plan to deploy network functions virtualization soon [1]

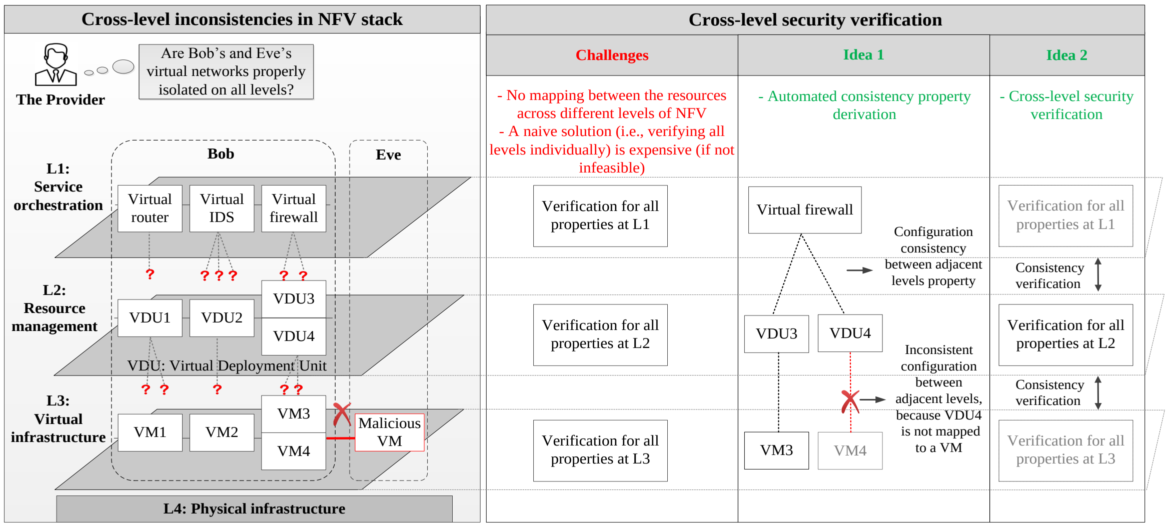

Cross-level inconsistencies in NFV stack
Are Bob's and Eve's virtual networks properly isolated on all levels?
The Provider
Bob
Eve
L1: Service orchestration
Virtual router
Virtual IDS
Virtual firewall
L2: Resource management
VDU1
VDU2
VDU3
VDU4
VDU: Virtual Deployment Unit
L3: Virtual infrastructure
VM1
VM2
VM3
VM4
Malicious VM
L4: Physical infrastructure
Cross-level security verification
Challenges
- No mapping between the resources across different levels of NFV
- A naive solution (i.e., verifying all levels individually) is expensive (if not infeasible)
Verification for all properties at L1
Verification for all properties at L2
Verification for all properties at L3
Idea 1
- Automated consistency property derivation
Virtual firewall
Configuration consistency between adjacent levels property
VDU3
VDU4
Inconsistent configuration between adjacent levels, because VDU4 is not mapped to a VM
VM3
VM4
Idea 2
- Cross-level security verification
Verification for all properties at L1
Consistency verification
Verification for all properties at L2
Consistency verification
Verification for all properties at L3

Figure 1: A motivating example illustrating the challenges of cross-level security verification in NFV and our ideas.

(e.g., L2 in this case), and then implicitly extend the result of such verification to other levels by verifying the consistency between adjacent levels.

To instantiate those ideas, we propose a security verification system, NFVGuard+, for the efficient and practical cross-level security verification of NFV stack. NFVGuard+ leverages formal methods to model the audit data and properties as a Constraint Satisfaction Problem (CSP) and employs the Sugar solver [22] to verify compliance. To facilitate this, we first create an Entity-Relationship (ER) model to systematically capture NFV entities and their relationships. Next, we identify consistency properties from the ER model and design an algorithm to automatically derive them from the model. We then develop our cross-level security verification approach, utilizing the ER model for data collection, processing, and formal verification. Finally, we demonstrate the applicability of our solution by integrating it into a real NFV testbed based on OpenStack/Tacker [23] and evaluate its efficiency through experiments with both real and synthetic data.

Security Capabilities of NFVGuard+. NFVGuard+ is designed to ensure the configuration of NFV stack complies with given security and consistency properties. Its results can provide either formal proof for such security compliance, or (in the case of non-compliance) counterexamples, i.e., policy violations in the NFV configuration. Although not specifically designed for attack detection, the policy violations identified by NFVGuard+ can potentially indicate the presence of misconfigurations, vulnerability exploitations, or other threats that have caused such policy violations, as long as these leave some traces in the logs or configuration. However, it is not designed to provide specific details about the underlying vulnerabilities (which requires vulnerability analysis) or attacks (which requires intrusion detection). Finally, it cannot detect policy violations leaving no traces, such as those caused by side-channel attacks or log tampering.

Comparison to Existing Solutions. In comparison to most existing NFV security verification solutions (e.g. [4]–[18], [24]), which primarily focus on a single level (mostly SFC), NFVGuard+ has a different focus, i.e., ensuring the security across all levels of the NFV stack. As we will demonstrate later in Section 2.3, this cannot be easily achieved using existing single-level solutions due to some unique challenges. Furthermore, although some approaches (e.g., [25], [26]) touch on the multilevel aspect of NFV, they do not formally model the verification problem as we do, cannot provide the same rigorous security proof provided by formal methods [25], or focus on a narrow scope of attacks (e.g., through VM placement optimization [26]). A detailed comparison is provided in Table 5.

In summary, our main contributions are:

- As per our knowledge, this is the first cross-level security verification system for NFV that supports automated identification of the properties and their data sources such that less human intervention would be needed.
- We are also the first to build an Entity Relationship (ER) model for NFV, which captures knowledge about the system entities and their relationships across different levels of the NFV stack. We provide a concrete guideline on how to effectively identify security properties utilizing the ER model. Those can be potentially useful for developing other security measures for NFV beyond security verification.
- We implement our solution and integrate it into a real NFV testbed built using OpenStack/Tacker [23] (a popular platform for deploying NFV [27]). Additionally, we experimentally evaluate our solution using both synthetic and real data (from one of the largest telecommunications vendors), demonstrating its efficiency and practicality (e.g., it took $\sim$2m to verify a large dataset of 100K VMs and we show that this time can be significantly reduced (to few seconds) by conducting the verification in parallel (more details are provided in Section 7)).

In the preliminary version of our work [28], we present the basic concept of cross-level verification for the NFV stack. This paper turns this concept into an efficient and practical system with the following major extensions. First, we re-design our approach to support the automated identification of properties (Section 3). Second, we propose a new *Entity-Relationship (ER)* model to capture NFV entities and their relationships across different levels of the NFV stack (Section 4.1). Third, we present a new approach for identifying consistency properties from the ER model and design an algorithm to automatically derive those properties from the model (Section 4.2). Fourth, we devise a new system that leverages our ER model for conducting each step of the cross-level verification, and we provide a concrete guideline for the user to utilize our ER model to efficiently identify new security properties (Section 5). Finally, we implement and integrate the system into our NFV testbed (Section 6) and conduct new experiments to evaluate its performance under different configuration scenarios (Section 7).

2 Preliminaries

This section provides the preliminaries.

2.1 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) [29]. Figure 2 illustrates the multilevel NFV deployment model [3] (on the right) with the mapping to a simplified view of the ETSI NFV reference architecture [29] (on the left). The NFV deployment model complements the ETSI NFV reference architecture with deployment details found in multiple open source platforms including Networking Automation Platform (ONAP) [30], Tacker [31], OpenStack [23]. Specifically, the deployment model depicts the NFV stack at four abstraction levels: *Service Orchestration (L1)* (which supports the specification, on-boarding, and lifecycle management of network services. Also, it could optionally include the SDN Orchestrator (SDNO) for the automated management of network resources and services), *Resource Management (L2)* (which supports the instantiation of network services and the management of computing, 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).

2.2 Security Properties for NFV

Security properties of NFV define the desired security states of the NFV deployment that are usually specified by the tenants and/or providers. Very often, these properties are inspired by security standards (e.g., ETSI [2] and ISO 27002 [32]) that outline fundamental security principles and recommendations for guiding the providers and for assisting the tenants in assessing the overall security compliance with the provider's NFV infrastructure. For this purpose, we conduct a study on the standards related to NFV (e.g., IETF-RFC7498 [33], and ETSI [2]), along with the standards related to various components of an NFV stack, such as cloud and SDN (e.g., ISO 27002 [32] and CCM [34]). Then we extract a list of security properties (the list can be found in the preliminary version of this work [28], which is omitted here due to space limitation) from those standards and the literature which can be used for the security verification for NFV. Please note that while this list is not meant to

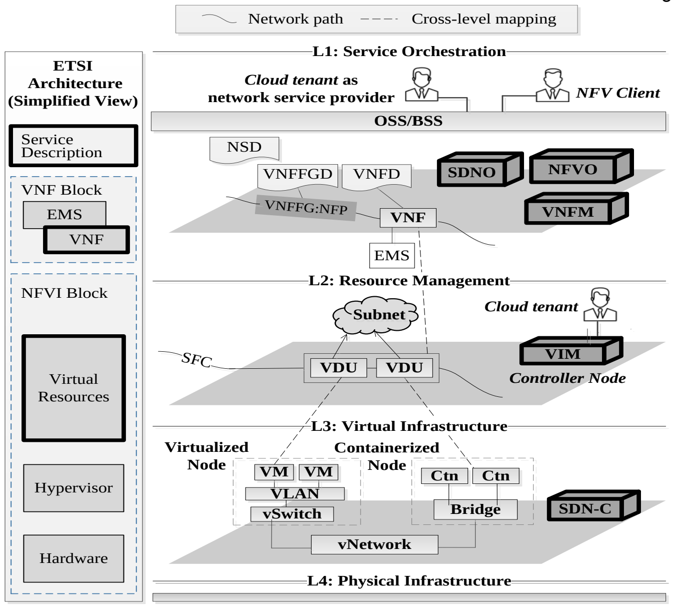


Figure 2: The multilevel NFV model [3].

be comprehensive, it can be easily extended to encompass additional security properties and even user-defined properties. Our approach can verify any security property as long as its expressed using formal methods. However, in this paper we focus on verifying the compliance of security properties related to the static configuration of the virtualized infrastructure, such as the proper configuration of isolation mechanisms. Dynamic properties, such as those related to reachability and network forwarding functionality, are beyond the scope of this paper and will be addressed in future work. To add specificity to our discussions, we provide a sample property, and subsequently, in Section 5, we show its verification process.

Example 1 ***Virtual resources isolation (no common ownership) property.*** *Aims at verifying that each virtual resource is exclusively owned by a single tenant unless specified by a user-defined policy. Specifically, in this paper we aim to verify that all VDUs composing a specific SFC at the management level are owned by a unique tenant, namely the owner of the SFC service.*

2.3 Challenges to Cross-Level Security Verification

Conducting security verification across different levels of NFV-stack (a.k.a. cross-level security verification for NFV) exhibits several unique challenges.

Identifying the NFV-Stack Entities and Their Relationships. To develop a cross-level verification system and identify the necessary input data for verifying various security properties, its essential to thoroughly understand the NFV system's design and workflow, which might be intractable as the NFV stack is a complex system with many interdependent entities located at different abstraction levels (as explained in Section 2.1). Moreover, the NFV standards (e.g., IETF-RFC7498 [33] and ETSI [2]) do not provide the necessary details for fully understanding the NFV system workflow and mapping the states of network services across the layers.

Locating the Data Sources for Security Properties. To verify a given security property, it's necessary to identify all relevant data sources and determine what data to collect from each source. This would require a good understanding of the property and accurately mapping its semantics to the corresponding NFV system resources, which also requires adequate awareness of the entities and their relationships within the NFV stack. Several security properties may require data from multiple levels of the NFV stack, depending on the involved data sources and their associated relationships. E.g., verifying SFC traffic isolation property [35], entails collecting data from the VDUs at L2, as well as from VMs and vSwitches at L3.

Data Correlation and Aggregation. Data sources are typically scattered across multiple physical servers and different NFV stack levels, each with its own data format (e.g., SFC traffic steering data is stored as OpenFlow rules at L3 and as database instances at L2). Therefore, its necessary to process the data into a consistent format and piece together related data within the same level (i.e., data aggregation), especially when audit data is scattered across different tables (e.g., the SFC data resides in different Neutron tables and Nova databases). Moreover, we need to link between data across different levels (i.e., data correlation) to obtain the necessary information for verification. These challenges will be addressed in Sections 4 and 5.

2.4 Threat Model

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 intents. Similar to most security verification solutions (e.g., [32], [36]), we trust the NFV provider for the integrity of the audit input data (e.g., logs and configurations). We also assume that the ER model correctly captures all the relations between the NFV system entities within the same level and captures all the mapping between cross-level entities, and any new changes in the system design affecting those relationships and mappings will be updated in the ER model. We assume that the properties defined in the paper are correct and complete i.e., it encompasses all the data and required relations to describe the given property. We also assume that one-level security property verification combined with verifying consistency properties for all levels would be sufficient for cross-level verification of a security property (as detailed in Section 4). Whereas, consistency property inspects whether the specifications set by the tenants or service providers are implemented correctly in the NFV system and that the implementation of resources at a specific level is instantiated correctly at the underlying level(s). This paper focuses on the verification of consistency properties and security properties related to the static configuration of the virtualized infrastructure, such as the proper configuration of isolation mechanisms. Any property violation that is not reflected on logs and configurations is beyond the scope of this paper. Although dynamic properties, such as reachability-related properties, also can be verified through formal methods (Lopes et al. [37]), these are out of the scope and they will be investigated in our future work.

Additionally, although our cross-level security verification solution can detect a violation of security properties, its not designed to attribute such a violation to underlying vulnerabilities (i.e., vulnerability analysis) or specific attacks (i.e., intrusion detection). However, mitigation solutions (e.g., [38], [39]) can be applied to address the risks associated with security breaches or vulnerabilities. These include security hardening options such as updating and patching vulnerabilities, enforcing strict security policies and access controls, conducting regular security audits, penetration testing, hypervisor introspection, remote attestation, and rollback to known good configuration.

3 Overview

Figure 3 shows an overview of NFVGuard+ including its three major steps and application to NFV.

1. Constructing the ER Model. To model the interconnectivity between different components in an NFV system, we construct the ER model that mainly captures: (i) the relationship between NFV entities within the same level, and (ii) the mapping between NFV entities from different levels (detailed in Section 4.1).

2. Automated Consistency Property Derivation. We automatically derive consistencies (which will be used in cross-level security verification later) between different entities in the NFV stack based on the ER model. More specifically, we derive properties that include the: (i) consistency of entity configurations, (ii) consistency of the relationship between two entities, and (iii) consistency of cross-level mapping (detailed in Section 4.2).

3. Cross-level Security Verification. We conduct cross-level security verification by utilizing two major steps: (i) verifying a security property for one level, and (ii) applying that verification result to other levels using the consistency results. We also provide a general guideline for the users to identify new properties (detailed in Section 5).

Application to Openstack/Tacker. As a potential application of our solution, we integrate NFVGuard+ with OpenStack/Tacker (a popular choice for NFV deployment) [23]. In our implementation, the user-defined network service descriptors are uploaded to Tacker through Horizon/CLI [23]. We choose the latest version of OpenStack (i.e., Rocky) and Tacker (i.e., Tacker-0.10.0) [23] to obtain the most recent features of NFV deployments. Finally, the traffic steering among the VNF elements is handled by the OvS switches [40]. We will detail the testbed data generation approach, report implementation challenges, and describe the integration of NFVGuard+ into the testbed in Section 6.

4 ER Model Construction and Consistency Property Identification

This section shows how the ER model is built and how the consistency properties are identified based on the model.

4.1 Constructing the Entity Relationship (ER) Model

To capture the relationships between NFV entities (e.g., between *VNFFG* and *Path* entities) both within and across NFV levels, we devise an ER model for the NFV stack (shown in Figure 4). The shaded nodes represent NFV-related entities, while non-shaded nodes represent entities related to the underlying infrastructure. The directed edges show the relationships between those entities at the same

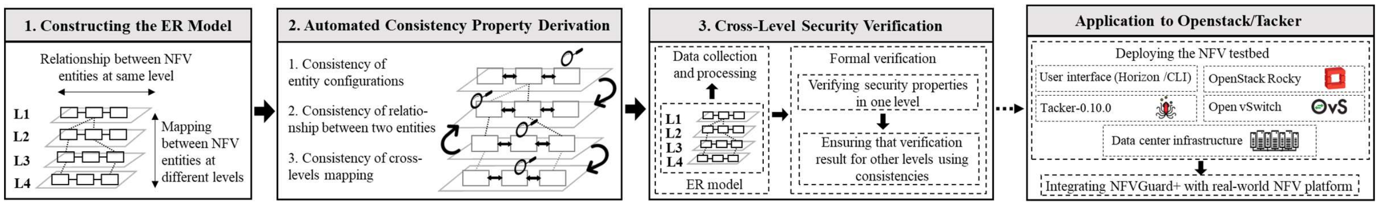

1. Constructing the ER Model
Relationship between NFV entities at same level
L1
L2
L3
L4
Mapping between NFV entities at different levels
2. Automated Consistency Property Derivation
1. Consistency of entity configurations
2. Consistency of relationship between two entities
3. Consistency of cross-levels mapping
3. Cross-Level Security Verification
Data collection and processing
ER model
Formal verification
Verifying security properties in one level
Ensuring that verification result for other levels using consistencies
Application to Openstack/Tacker
Deploying the NFV testbed
User interface (Horizon /CLI)
OpenStack Rocky
Tacker-0.10.0
Open vSwitch
Data center infrastructure
Integrating NFVGuard+ with real-world NFV platform

Figure 3: An overview of the NFVGuard+ approach.

level, while (1:1), (1:M), (M:1), and (M:M) represent the corresponding cardinalities of the relations. The dashed line edges represent the cross-level mapping between the entities at different levels, which have a (1:1) cardinality.

We construct the model by performing a comprehensive study of the system configurations of a real NFV testbed implemented using OpenStack/Tacker [23] (detailed in Section 6), and relevant literature on modeling and deploying NFV and virtualized infrastructures (e.g., [3], [41], [42]). We further validate our model with several industrial experts on NFV from a large telecommunication vendor. In the following, we elaborate on our ER model construction process.

Constructing the Nodes of the ER Model. According to the NFV deployment model (discussed in Section 2.1), we divide the ER model into four levels, the Service orchestration level (L1), resource management level (L2), virtual infrastructure level (L3), and physical infrastructure level (L4). Then, to capture the system entities at each level, we study the deployment details of NFV environments [3], [23] and the supporting technologies (such as network virtualization technologies like VLAN and VXLAN [43]) for implementing the NFV. Then, we represent the identified entities as the ER model nodes.

Example 2 *In this example, we identify the* `NS provider`*,* `NSD`*, and* `NS` *nodes in the ER model. The NS provider uploads the Network Service Descriptors (NSDs) at the Service Orchestration level (L1), which define the network service based on user requirements. Each NSD creates one or more NSs, stored as entities in the NFV system, along with the NS provider's ID and information. Thus, the NS provider, NSD, and NS entities are represented as nodes in the ER model, as shown in Figure 4 at L1.*

Constructing the Edges of ER Model. We construct the ER model with two types of edges based on the: (i) relationships between NFV entities at the same level, and (ii) mappings between entities from different levels. In particular, we identify the relationships and constraints among same-level entities and represent them as directed edges with cardinality attributes. Additionally, some system entities at one level are implemented as different entities at the next level. The relationships between these entities can be utilized for verification. We represent these relationships as cross-level mapping edges connecting NFV entities across different levels.

Example 3 *Since the NSD creates the NS (as explained in Example 2), we establish a directed edge between these entities, labeled CreatedFrom(M:1)(Figure 4 at L1), to represent their relationship. The cardinality* ((M:1)) *reflects the constraints governing this relationship: the NSD can create multiple NSs, but each created NS belongs to only one NSD template. Furthermore, VNF specifications at L1 of the NFV system are instantiated as VDUs at L2. Thus, we represent this relationship as a cross-level mapping between the VNF and VDU entities.*

4.2 Automated Consistency Property Derivation

This section illustrates how the ER model is utilized to automatically derive consistency properties (which will be used later for our cross-level verification in Section 5).

The relationships between entities in the ER model reflect fixed configuration constraints within the NFV system. For instance, the relationship between the *Path* and *Chain* entities at L1 (refer to Figure 4) is (1:1), indicating that each *Chain* is linked to a specific *Path*, and each *Path* corresponds to one *Chain*. Deviating from these fixed configurations can lead to unintended service behavior or interruptions.

Accordingly, we can derive properties, namely consistency properties, to verify whether any instances created within the NFV system comply with the established configurations. These properties can be automatically obtained by systematically parsing the entities and edges of the ER model, with the assumption that the model correctly captures all relationships between the NFV system entities at both the same and across different levels (Section 2.4). In particular, we can derive the following consistency properties.

Consistency of Entity Configuration. Each node in the ER model represents a system entity with various configuration options determined by the specifications provided by NFV tenants. We explore each node to derive a corresponding consistency property that ensures the alignment of entity configurations with the defined specifications.

Consistency of Relationships Between Entities at the Same Level. The directed edges between two entities at the same level in the ER model signify their relationship, reflecting system configurations and tenant specifications. We explore these edges to derive consistency properties that ensure the relationships align with both system configurations and tenant specifications.

Consistency of Relationships Between Entities Across Different Levels. Similarly, the dashed line edges between two entities in the ER model across adjacent levels represent a relationship between them. Specifically, this indicates that an entity at a higher level must have a corresponding implementation at the next level. We explore these edges to derive consistency properties that ensure the integrity of the mappings. Table 1 presents an excerpt of consistency properties automatically derived from the ER model, including their corresponding sources, and descriptions.

The aforementioned consistency properties can be automatically obtained from the ER model by representing it as a graph. In this graph, entities are depicted as nodes,

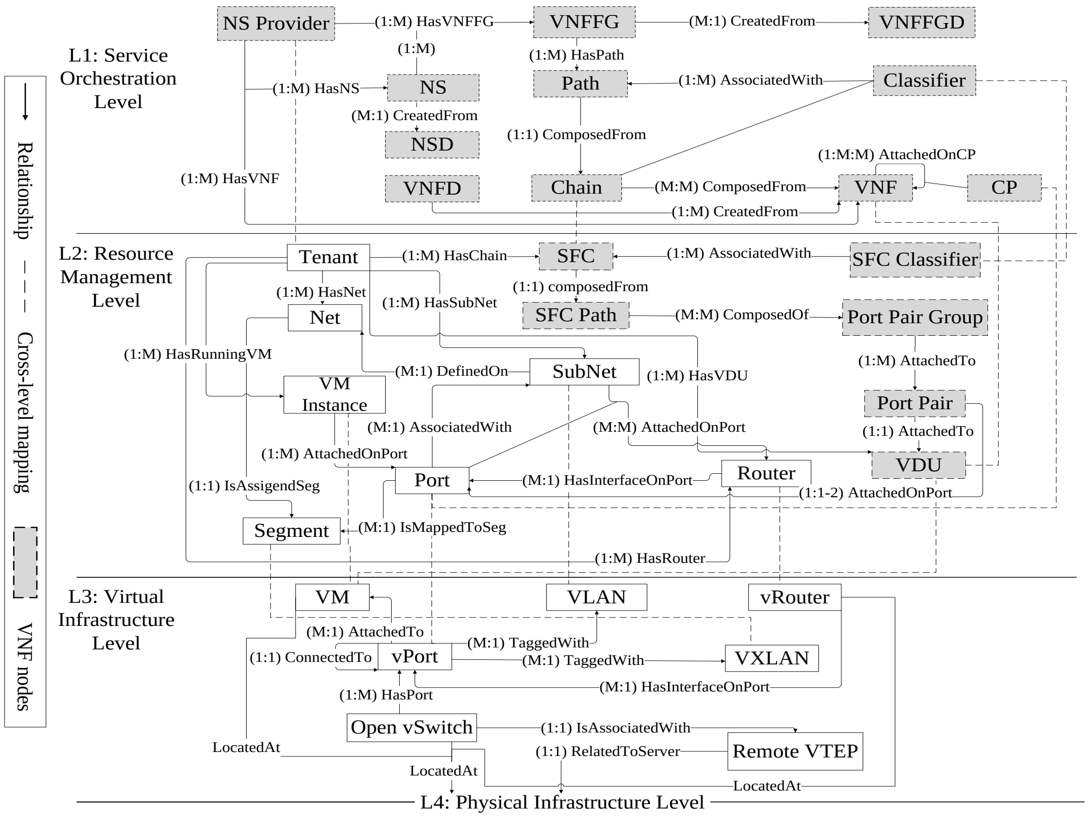


Figure 4: The ER model of the NFV stack.

Property	ER model source	Description
Classifier integrity	L1: *Classifier* entity	Classifier configurations should be consistent with tenant-defined specifications
Forwarding correctness	L1: *AssociatedWith* relationship between the *Classifier* and *Path* entities	The classifier should be associated with the correct path as outlined in the tenant specif-ications to ensure accurate traffic steering
Service chain configuration consistency	L1, L2: Cross-level mapping between the *Chain* and *SFC* entities	Service chain created at L1 should be correctly instantiated as SFC at L2

Table 1: Example of consistency properties identified from the ER model entities and relationships.

and relationships are depicted as directed edges, attributed with relationships and their cardinalities. By traversing the graph, each node and its connected edges are processed to extract the relevant consistency properties. These properties are then stored in two lists: *EntityConsistencyProperty* for node-level consistency and *EdgeConsistencyProperty* for relationship-level consistency.

5 Cross-Level Security Verification

This section describes how NFVGuard+ conducts cross-level security verification.

Data Collection. To conduct cross-level security verification in NFV, data must be collected from various sources across different levels of the NFV stack. For example, to verify whether a *VNFFG* is correctly implemented according to the specification, we need to collect data from various levels, including the VNFFG specification from the Tacker database at L1, data about VDUs and ports from the Nova and Neutron databases at L2, and the OpenFlow rules at L3 from multiple servers. Typically, this would involve manually inspecting the configurations at each level to identify relevant data for each property. However, by utilizing the ER model, we can efficiently identify the necessary data for each property as follows.

First of all, we must identify the property requirements (what needs to be verified) and determine their scope (which level they pertain to). Next, we will map these requirements at each level to the ER model and identify the entities within the model that relate to the property. For instance, the *VNFFG configuration consistency between L1/L2* property (refer to [28]) requires that the VNFFG design (at L1)-including the size of the VNFFG, the VNF sequences, and the classifiers definition-be correctly instantiated into corresponding SFC configurations (at L2), including the SFC size, VDU sequences, and classifier details. One of the requirements for this property at L1 is to determine the size of the VNFFG, which indicates the number of VNFs that comprise it. By referencing the ER model at L1, we should relate this requirement with the corresponding entities at this level. Since it pertains to the VNFFGs, we will select the *VNFFG* entity as a relevant entity for this property. Then, we will examine the relationships associated with this entity to check if they can be utilized by the property. For example, by examining the relationships associated with the *VNFFG* entity, we can observe that the *VNFFG* may consist of one or more paths, with each path comprising a chain of VNFs. This highlights the importance of the *Path*, *Chain*, and *VNF* entities in determining the size of the VNFFG.

Afterward, we will collect the relevant data for the property based on the identified entities and by consulting the data sources table (Table 2), which is created in conjunction with the ER model. For example, the ID of each VNFFG is stored in the *VNFFG* entity along with the ID(s) of the path(s) it is composed of, and each *Path* entity includes the ID of the chain that makes it up. While, the size of the VNFFG, can be determined from the data source of the `Chain` entity, as indicated in the *Description* column of Table 2. Likewise, the relevant data for the other requirements of the property are identified in the same manner.

Entity	Data source	Description
Chain	The `vnffgchains` table in Tacker database	Identifies the CPs and the VNFs in the chain and the sequential order of the VNFs as outlined in the specifications
Classifier	The `vnffgclassifiers` table in Tacker database	Identifies the classified traffic flows entering the VNF chain path, typically including details like source port and IP protocol
SFC	The `sfc_port_chains` table in Neutron database	Stores the ID of the chain created at L1 and document its instantiation and specifications, including the flow order between VNFs
Open vSwitch (OVS)	`ovs-fields` in OVS and OpenFlow tables	Store information related to the forwarding behavior of the network services

Table 2: An excerpt of the data sources for some of the entities in the ER model, along with a description of the types of data they contain.

Data Processing. The data required to verify a specific security property could be collected from multiple levels of the NFV stack and it may differ in format, as each level employs distinct technologies (such as resource management at L2 and virtual networking elements at L3) and stores data in different formats (e.g., SFC traffic steering is stored as OpenFlow rules at L3 and as database entries at L2). Moreover, this data could be scattered (e.g., across different database tables or different OvSs) and might not directly reflect the necessary information needed for verification. Therefore, we process the collected data to generate meaningful information for verification and ensure its in a consistent format compatible with the formal verification engine (e.g., the input format for the Sugar CSP solver). The processing of the collected data is outlined below.

1) Data correlation: Due to the distributed nature of the audit data (e.g., data may be scattered across different services at the same level, such as Nova or Neutron in OpenStack or among physical servers), we need to correlate the collected data within each level to produce meaningful information for verification [41]. For example, *VNFFG_1* is implemented at L3 as three VMs (*VM_01*, *VM_02*, and *VM_03*) hosted on two physical servers. To verify the forwarding correctness of *VNFFG_1*, we need to collect the flow rules (determines how traffic flows through these VMs) stored on both physical servers and scattered across multiple tables on each server. For example, if *VM_02* and *VM_03* are on the same physical server and we want to verify whether *VM_02* is forwarding traffic to *VM_03*, we will need to examine the flow rules stored in tables 0, 5, 10, and the Group table. Therefore, we need to correlate all these data to piece together sufficient information for verification. The relationships between system entities at each level of the ER model help to identify the data that needs to be correlated. For instance, the relationship between the *VNFFG* entity and the *Path* entity indicates that they are interconnected and their data could be correlated.
2) Data aggregation: Audit data for specific properties, such as consistency properties, could be distributed across different levels of the NFV stack. Therefore, we must aggregate the data from these different levels to compile sufficient information for verification. For example, to verify whether a VNFFG is correctly implemented according to the specification, we need to collect the specification data at L1, aggregate it with the instantiation data at L2, and further combine it with the implementation data at L3. The cross-level mapping relationships between system entities at each level of the ER model assist in identifying the data that needs to be aggregated.

Formal Verification. We propose to apply formal methods to verify the compliance of the NFV stack against the identified security and consistency properties. In this work, we formalize the properties as a Constraint Satisfaction Problem (CSP), a time-proven technique for expressing many complex problems. We then apply Sugar [22], a well-established constraint solver, to check whether these properties are satisfied. We detail the verification process as follows.

To systematically verify the NFV-related properties, we need to transform the property requirements as well as the involved ER model entities and their instances (i.e., the system data) into the corresponding CSP code. The CSP code mainly consists of four parts:

- Variable and domain declaration. Entities of the ER model are expressed as CSP variables with their domain definitions (over integer), where the domain encompasses all instances defined by the system data. For example, for the VNFFG configuration consistency between L1/L2 property, the VNFFG entity (refer to Figure 5) is expressed as the variable *fg* defined over the domain *VNFFG* such that `(domain VNFFG 0 max_vnffgs)` is a declaration of a CSP finite domain of VNFFGs, where each value between `0` and `max_vnffgs` is for a corresponding data instance in the NFV system.
- Relation declaration. The ER model relations, involved in the property requirements, are converted into CSP relations over variables with a support consisting of tuples of system data. For example, the relation between the VNFFG and its path (refer to Figure 5) is defined as the CSP relation `(relation HasPath 2 (supports(fg path))`, where instances of a given relation are the set of tuples corresponding to the entities instances. The CSP relations describe the current state of the system.
- Constraint declaration. We define constraints, in terms of CSP predicates, over the involved relation to specify the conditions that the instances of these relations should meet. Since CSP solvers provide solutions only in case the constraint is satisfied (SAT), we define constraints using the negative form of the property to obtain a counter-example in case of a violation.
- Body. We combine different predicates based on the properties to verify using Boolean operators.

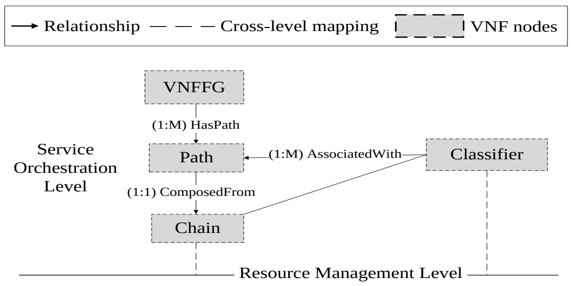


Figure 5: Thumbnail of the ER model showing entities for verifying VNFFG configuration consistency property at L1.

When the CSP solver (i.e., Sugar) solves the constraints and finds no solution (UNSAT), the verified properties are reported to be compliant. Otherwise, the solution provided by the CSP solver gives the variables' instances for which the negative form of the property is satisfied, meaning that a violation has occurred. For instance, we express the property virtual resource isolation presented in Example 1 using the following CSP relations. `HasChain(t, sfc)` which evaluates to true if tenant *t* has/owns a running SFC *sfc*, `SFCHasVDUs(sfc, vdu)` which evaluates to true if the SFC *sfc* has assigned VDU *vdu*, `HasVDU(t, vdu)` which evaluates to true if the tenant *t* has a running VDU *vdu*. Then we define the negation of the property in terms of a predicate over those relations to obtain a counter-example in case of a violation, shown as the `VirtualResourceIsolation` predicate in Listing 1 (an excerpt of Sugar code). Example 4 shows how Sugar verifies this property and allows for obtaining the violation evidence.

Example 4 *Suppose that a tenant t with the* `Tenant_ID` *(18e552) is encoded as (10) in listing 1, the* `Chain` *(3cf7ca68) he owns as (1), and the VDUs (49ce0b1e, 738bb405) as (15, 16), respectively. The predicate* `VirtualResourceIsolation` *will evaluate to true if any of the VDUs assigned to the chain (1) that belongs to tenant (10) is owned by another tenant. According to the relation instance* `HasVDU(11 16)`*, the chain (1) has a VDU (16) that does not belong to tenant (10). Therefore, the predicate evaluates true and the output of Sugar code is (SAT) with evidence about what values breached the property i.e., (t=10; sfc=1; VDU1=16).*

```
//Domains and variables declaration
(domain TENANT 0 10,000) (domain SFC 0 5000)
(domain  VDU 0  100,000)
(int t TENANT) (int sfc SFC) (int vdu VDU)
//Relations Declarations
(relation HasChain 2(supports((10 1)(12 3)))
(relation SFCHasVDUs 2(supports((1 15)(1 16)))
(relation HasVDU 2(supports((10 15)(11 16)))
//Predicate Declaration
(predicate(VirtualResourceIsolation t sfc vdu)
(and (HasChain t sfc)(SFCHasVDUs sfc vdu)(not(
    HasVDU t vdu))))
//The Body
(VirtualResourceIsolation t sfc vdu)
```

Listing 1: An excerpt of Sugar source code.

After verifying the NFV-related properties, we ensure the verification result for other levels using consistencies. The consistency between different levels of the NFV stack can be utilized to improve the performance of the verification (as we illustrate in the motivating example in Section 1). The key idea is to leverage the consistency result to perform security verification at one level of the NFV stack, instead of verifying the same security property at each level separately. As long as the NFV stack levels are consistent, the verification results at one level would be applicable to other levels. We show the performance improvement that we gain by utilizing the consistency property in experiments (Section 7).

6 Application to OpenStack/Tacker

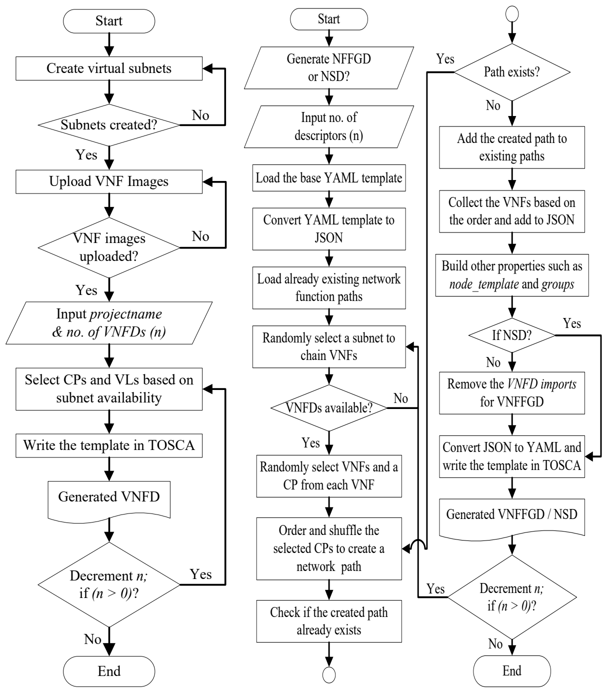
In this section, we detail the deployment and data generation of our NFV testbed, discuss the challenges encountered during this process, and detail the implementation of NFV-Guard+.

6.1 Deploying the NFV Testbed

NFV Testbed Implementation. We build our NFV testbed using OpenStack [23] with Tacker [31] due to its growing popularity in the real world (e.g., [27]). More specifically, we rely on OpenStack for the Virtual Infrastructure Manager (VIM), which has been adopted by 96% of CSPs and more than 60% of the telecom operators in their NFV deployments [44]. We rely on Tacker, an official OpenStack project, for both VNFM and NFVO modules based on the ETSI MANO architectural framework [29]. We choose the latest version, i.e., OpenStack Rocky and Tacker-0.10.0 [23] to obtain the most recent features of NFV deployments.

NFV Data Generation. We intend to deploy a large-scale NFV system to assess the performance of NFVGuard+. However, to the best of our knowledge, there is no publicly available dataset of TOSCA [45] deployment descriptors for a large-scale NFV deployment. Therefore, we develop Python scripts to generate various Virtual Network Function Descriptors (VNFDs) and Virtual Network Function Forwarding Graph Descriptors (VNFFGDs) in TOSCA, and we onboard those to our testbed to deploy different network services and generate large-scale NFV datasets. To ensure more diversity, we randomly choose a few parameters in the template while generating the deployment descriptors: 1) the number of network ports per VNF, 2) the number of VDUs per VNF, 3) the Flavor for each VNF and VDU, 4) the number of VNFs for each Network Function Path (NFP), 5) the order of VNFs for each NFP, 6) the flow-classifier criteria for each NFP, and 7) the number of NFPs for each VNFFG.

Specifically, the scripts first generate a diverse set of VNFDs for a given tenant by customizing a base template. After that, they generate multiple VNFFGDs (resp. NSDs) by creating unique network function paths using the available VNFDs. Then, these descriptors are onboarded to the VNFM and the NFVO modules in Tacker, respectively, through Horizon/CLI [23]. Once onboarded, the TOSCA templates are interpreted and translated to Heat templates [23]. Then, using the Heat template, Tacker leverages Nova to provision the virtual instances implementing the VNFs, and Neutron to provision the virtual networks that provide the connectivity to and from each VNF. Finally, the traffic steering among the chains of VNFs is handled by the OvS switches [40]. Figure 6 shows the detailed flowcharts for generating VNFDs and VNFFGDs.


Start
Create virtual subnets
Subnets created?
No
Yes
Upload VNF Images
VNF images uploaded?
No
Yes
Input projectname & no. of VNFDs (n)
Select CPs and VLs based on subnet availability
Write the template in TOSCA
Generated VNFD
Decrement n; if (n > 0)?
Yes
No
End
Start
Generate NFFGD or NSD?
Input no. of descriptors (n)
Load the base YAML template
Convert YAML template to JSON
Load already existing network function paths
Randomly select a subnet to chain VNFs
VNFDs available?
No
Yes
Randomly select VNFs and a CP from each VNF
Order and shuffle the selected CPs to create a network path
Check if the created path already exists
Path exists?
Yes
No
Add the created path to existing paths
Collect the VNFs based on the order and add to JSON
Build other properties such as node_template and groups
If NSD?
Yes
No
Remove the VNFD imports for VNFFGD
Convert JSON to YAML and write the template in TOSCA
Generated VNFFGD / NSD
Decrement n; if (n > 0)?
Yes
No
End

(a) VNFD Generation (b) VNFFGD/NSD Generation

Figure 6: The process of generating VNF and VNFFG/NS TOSCA template descriptors.

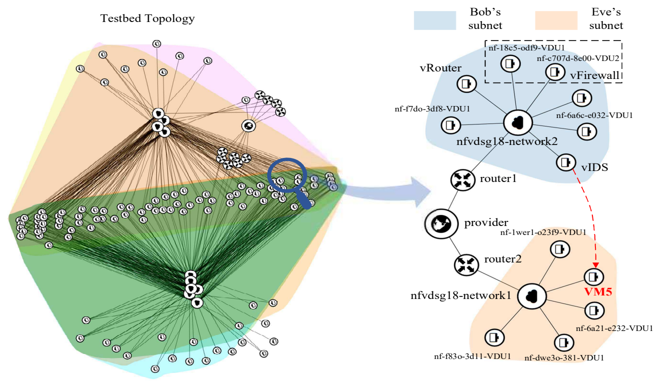


Figure 7: The topology of our NFV testbed (left) consisting of 20 tenants, 200 VNFFGs, and 200 VNFs and detailed view (in Horizon [23]) of an attack scenario similar to the motivating example in Section 1 (right).

VNFDs Generation. Figure 6(a) depicts the procedure to generate multiple VNFDs for a given tenant. Each VNFD is used to create one or several VNFs of the same type. More specifically, we use a base template that our generator customizes to create a diversified set of VNFDs. First, a set of virtual subnets is created, and then the corresponding VNF images are uploaded to be used within the VNFD templates. For each VNFD to be created, a set of subnets is selected, and then identifiers for connection points (CPs) and virtual links (VLs) associated with the VNF are created accordingly. Then, these identifiers are applied to fill in the VNFD template. Once these VNFDs are generated, they are used to generate VNFFGDs/NSDs.

VNFFGDs/NSDs Generation. Figure 6(b) depicts the process for generating multiple VNFFGDs (resp. NSDs) for a given tenant. Similar to generating VNFDs, we use a base VNFFGD (resp. NSD) TOSCA template with all the necessary attributes that our generator modifies accordingly to create a diversified set of VNFFGDs (resp. NSDs). The generator considers the number of VNFFGDs (resp. NSD), the available VNFDs, and the VNFFGD/NSD base template as its inputs. The VNFFGD/NSD base template (originally in YAML format) is first converted into JSON format for easier modification. Then a list of existing VNFFGD (resp. NSDs) is loaded and checked to avoid the creation of duplicates. To build new VNFFGDs/NSDs, a subnet is first randomly selected, and then the CPs connected to this subnet are collected from a random number of VNFDs. After that, these CPs (each represents a VNF) are shuffled first and then ordered to create a network function path. Once a path is created, its verified against the list of existing paths to avoid any duplication. Then, the VNFs are collected based on the order of CPs followed by the creation of other additional information such as `node_template`, `groups`, and `network_src_port_id` to complete the VNFFGD (resp. NSD). To finalize the generation of VNFFGD (resp. NSD), the JSON is converted into YAML again and then saved as a TOSCA template file.

Figure 7 (left) is generated using OpenStack Horizon [23] to provide an overview of the network topology of our NFV testbed consisting of 20 tenants, 200 VNFFGs (each VNFFG consists of 10 VNFs), and each tenant has 10 VNFFGs. The figure shows the interconnections between the provider network and different tenant subnets (which are highlighted in different colors for each tenant) with their corresponding routers and VNFs. Figure 7 (right) shows a detailed view of an attack scenario similar to the motivating example (Section 1) where a malicious virtual machine (`VM5`) from the network of *Eve* (`nfvdsg18-network1`, highlighted in orange), is stealthily added to the service function chain of *Bob* implemented in his subnet (`nfvdsg18-network2`, highlighted in blue).

NFV Testbed Implementation and Data Generation Challenges. Hereafter, we will discuss the implementation and data generation challenges, causes of failures, and our solutions. Due to space constraints, not all challenges are covered here.

Version Mismatch. Basic NFV implementation with OpenStack requires careful orchestration of at least 14 OpenStack services. Version mismatch among these services can lead to deployment failures and pose significant troubleshooting challenges. For instance, we encountered a silent failure in OpenFlow rules update, due to a version mismatch between Neutron and OvS. We addressed this by downgrading Neutron version.

Manual Effort. During the installation process, we encountered an unexpected freeze. To bypass the freeze and complete the installation, we manually installed some services specifically, Mistral and Tacker.

Undocumented Deployment Constraints. During data generation, we encountered VNFFG creation failures due to undocumented deployment constraints within the VNFFG template. These failures involved the inability to chain VNFs using management ports or from different subnets, and required traffic to originate from the same subnet. We address these failures by VNFFG template validation.

6.2 NFVGuard+ Implementation

The data collection component is implemented to collect data from different OpenStack services, such as Tacker, Nova, and Neutron [23], as well as from the instances running on every compute node of OvSs. Specifically, we rely on the Tacker database to retrieve user-defined descriptors uploaded to the VNFM and NFVO modules of Tacker (e.g., VNFD and VNFFGD) as the basis for verifying most of the properties. We also rely on a collection of OpenStack databases, such as Neutron database for information about SFC networking (e.g., the sequence of service functions, the traffic steering in-between, and the traffic classifier) and Nova databases (e.g., 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., to check for properties such as inconsistencies between L2 and L3. To process the collected data, we implement the data processing component in Python and Bash scripts. 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 (e.g., the support of the relation *BelongsTo*) or recovered after data correlation. Second, our processing component formats each group of data as an n-tuple, e.g., (resource, tenant), (OVS, VLAN, VXLAN), etc. Finally, it uses the n-tuples to generate part of the Sugar [22] source code and appends the n-tuples with the variable declarations, relationships, and predicates for each security property. Then we develop a customized script 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 CSP solver version 2.3.3 [22]. Sugar then produces the verification results to either state the property holds or provide evidence when the property is breached.

7 EXPERIMENTS

This section evaluates the effectiveness of NFVGuard+ in terms of accuracy, efficiency, and scalability through experiments using real and synthetic datasets. In the following, we describe our experimental settings and findings.

7.1 Experiments with Synthetic Data

Experimental Settings. We deploy our testbed 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 NFVGuard+, we generate various synthetic datasets of different sizes varying from 1K up to 5K VNFFGs (representing reasonably large NFV setups [46]), and from 20K to 100K VMs. All data processing and experiments are conducted on the SuperServer with the verification tool, Sugar V2.3.3 [22]. Each experiment is performed 1,000 times to avoid any fluctuation caused by other operations on the server. The reported results show the efficiency and scalability of NFVGuard+.

Effectiveness Evaluation of NFVGuard+. To evaluate the effectiveness of our approach, we apply NFVGuard+ to pre-validated instances of security properties and assess its accuracy in verifying those instances. Table 3 shows some example security properties, their investigated instances, the instantiated Sugar code for each instance, and the corresponding Sugar output.

The accuracy of our approach depends on the precision of the formal verifier, specifically the Sugar SAT solver. To test the solver's accuracy, we provide the solver with pre-validated instances and compare its results with our own. In particular, we verify instances of the *VNFFG configuration consistency*, *virtual resource isolation*, and *mapping unicity VLANs-VXLANs* properties. These instances are first tested by us for compliance before being given to the solver. Then, we check if the solver incorrectly identifies any of the compliant instances as non-compliant. Our evaluation shows that the solver output is accurate, correctly identifying all instances as compliant. Examples of these instances are shown in the first four rows of Table 3, where the solver output is UNSAT, indicating that the instances comply with the corresponding properties. For clarity, the instantiated Sugar code has been shortened and simplified. For more details on Sugar syntax and the full code excerpt, refer to Section 4.

Next, we inject security breaches at different levels of the NFV stack and test the accuracy of our approach in identifying those breaches. First, by exploiting a privilege escalation vulnerability in OpenStack (OSSA-2017-004 [47]), we would be able to modify the specification of an SFC and add an additional VNF. Such a modification at L2 will not be reflected at L1, resulting in a breach of *configuration consistency between L1/L2* property. An instance of this breach is presented in Table 3. In verifying the property, the solver aims to identify any VNFs that are defined in L1 for the given chain but not in L2, and vice versa. The solver successfully identifies this breach and returns the values that cause the property violation, specifically: (VNFFG path: 20fp, SFC: 20fp, VDU2: 11f, VDU3: 12f) (refer to Table 3).

Second, we target the flow tables at L3 to create inconsistencies with higher levels. By triggering a virtual switch reconciliation during a network topology update, outdated flow rules are reinstalled, causing traffic to be steered according to old definitions [3]. This leads to a breach in configuration consistency between L3 and upper levels. An instance of this breach is presented in Table 3, where we assume the *configuration consistency between L1/L2* property was verified to be met by the configuration.

In particular, a VNFFG that initially forwarded traffic from a vRtr to a vFW is updated to route traffic from the vRtr to the vFW and then to an additional vDPI. While the SFC at L2 is updated, the L3 flow rules remain unchanged due to the virtual switch reconciliation vulnerability. In verifying the property, the solver aims to identify any discrepancies in the traffic steering information collected from the different levels. The solver successfully identifies this breach and returns the values (SFC: 30fp, Chain: 30fp, vFW: 18f, vDPI: 19f) as evidence of the violation. Additionally, we generate misconfigurations to create further breach instances. The last two rows of Table 3 provide examples of these instances.

Our tests demonstrate the effectiveness of our approach in providing accurate results for the specified security properties. In general, using formal methods in security verification is known to provide provably accurate results

Property	Property instance	Instantiated Sugar code	Sugar output
VNFFG configuration consistency between L1/L2	L1: (VNFFG_path: 10fp, VNF1: 4f, VNF2: 5f) and L2: (SFC: 10fp, VDU1: 4f, VDU2: 5f)	((predicate VNFFGConsistencyL1/L2) (and (L1Chain (10fp, 4f, 5f)) (L2Chain(10fp, 4f, 5f)) (10fp = 10fp)))	UNSAT
VNFFG configuration consistency between L2/L3	L2: (SFC: 10fp, VDU1: 4f, VDU2: 5f) and L3: (Chain: 10fp, VM1: 4f, VM2: 5f)	((predicate VNFFGConsistencyL2/L3) (and (L2Chain (10fp, 4f, 5f)) (L3Chain (10fp, 4f, 5f)) (10fp = 10fp)))	UNSAT
Virtual resource isolation	L2: (Tenant: 1t, SFC: 10fp, VDU1: 4f, VDU2: 5f)	((predicate VirtualResourceIsolation) (and (HasChain (1t, 10fp)) (SFCHasVDUs (10fp, 4f) (10fp, 5f)) (not(HasVDU (1t, 4f) (1t, 5f)))))	UNSAT
Mapping unicity VLANs -VXLANs	L3: (Port: 9p, Switch: 1s, VLAN: 7l, VXLAN: 10xl)	((predicate MappingUnicity) (and (AssignedVLAN (1s, 9p, 7l)) (MappedToVXLAN (1s, 7l, 10xl)) (MappedToVXLAN (1s, 7l, 10xl)) (not (10xl = 10xl))))	UNSAT
VNFFG configuration consistency between L1/L2	L1: (VNFFG_path: 20fp, VNF1: 10f, VNF2: 11f) and L2: (SFC: 20fp, VDU1: 10f, VDU2: 11f, VDU3: 12f)	((predicate VNFFGConsistencyL1/L2) (and (L1Chain (20fp, 10f, 11f)) (L2Chain (20fp, 10f, 11f), (20fp, 11f, 12f)) (20fp = 20fp)))	SAT: (VNFFG_path: 20fp, SFC: 20fp, VDU2: 11f, VDU3: 12f)
VNFFG configuration consistency between L2/L3	L2: (SFC: 30fp, vRtr: 17f, vFW: 18f, vDPI: 19f) and L3: (Chain: 30fp, vRtr: 17f, vFW: 18f)	((predicate VNFFGConsistencyL2/L3) (and (L2Chain (30fp, 17f, 18f), (30fp, 18f, 19f)) (L3Chain (30fp, 17f, 18f)) (30fp = 30fp)))	SAT: (SFC: 30fp, Chain: 30fp, vFW: 18f, vDPI: 19f)
Virtual resource isolation	L2: (Tenant: 1t, SFC: 10fp, VDU1: 4f, VDU2: 5f)	((predicate VirtualResourceIsolation) (and (HasChain (1t, 10fp)) (SFCHasVDUs (10fp, 4f) (10fp, 5f)) (not(HasVDU (1t, 4f) (2t, 5f)))))	SAT: (Tenant: 1t, SFC: 10fp, VDU2: 5f)
Mapping unicity VLANs -VXLANs	L3: (Port: 9p, Switch: 1s, VLAN: 7l, VXLAN: 10xl, VXLAN: 15xl)	((predicate MappingUnicity) (and (AssignedVLAN (1s, 9p, 7l)) (MappedToVXLAN (1s, 7l, 10xl)) (MappedToVXLAN (1s, 7l, 15xl)) (not (10xl = 15xl))))	SAT: (Port: 9p, Switch: 1s, VLAN: 7l, VXLAN: 10xl, VXLAN: 15xl)

Table 3: Example property instances for evaluating the effectiveness of NFVGuard+.

[48], [49] for given security properties. A practical challenge is for administrators to properly identify and define the security properties based on their specific needs. One potential solution to this challenge is to automatically extract security properties from standards using natural language processing (NLP) [50], [51], though this falls beyond the scope of this paper.

Efficiency of Verifying the Consistency Properties. In this experiment, we evaluate the efficiency (in terms of response time, CPU usage, and memory consumption) of NFVGuard+ in verifying the consistency properties derived from the ER model (refer to Section 4.2). We verify the *classifier integrity*, *forwarding correctness*, and *service chain configuration consistency* properties, which correspond to the consistency properties derived from the different objects of the ER model, i.e., node, edge, and cross-level edge, respectively.

According to Figure 8, the verification time increases almost linearly with the increased number of resources and the verification requires less than 1.5 seconds for all three properties even for the largest dataset. The verification of *service chain configuration consistency* property incurs the lowest response time, CPU, and memory consumption as shown in Figure 8 due to its simplest predicate with a smaller number of variables than the other two properties. This is expected as complex properties with a higher number of relations and variables generally take more time to process, and consume more memory and CPU. However, the maximum amount of CPU consumption is less than 12%, while the maximum memory consumption is only 1%. Hence, though the verification for the *forwarding correctness* property takes more time and consumes more resources than the *classifier integrity* property, the consumed resource still stays reasonably low.

Efficiency of Cross-Level Security Verification. In this set of experiments, we evaluate the verification time required by the candidate properties presented under different configuration scenarios. More specifically, the first configuration scenario assumes that the NFV configuration has no violation of any of the considered properties (detailed later in this section and depicted in Figures 9 (left) and 10 (left)), while in the second scenario (detailed later in this section), we inject several violating instances for each of the tested properties and consider the time to report the evidence only for the first breach (Figures 9 (middle) and (right) and 10 (middle)), in case a fast binary answer on the compliance status of the system is required by the system administrator/auditor. We then consider the average response time to find all compliance breaches (detailed later in this section and Figures 10 (right), 11 and 12 (left)). For each of the investigated scenarios, we consider the consistency properties, *VNFFG configuration consistency between L1 and L2*, and the *VNFFG configuration consistency between L2 and L3*. We also consider the security properties, *virtual resource isolation* (L2), and the *mapping unicity VLANs-VXLANs* (L3).

Note the required time for detecting non-compliance with the consistency properties also depends on the level where the breach is detected. For instance, if we detect a violation in the consistency property at L1/L2, then the verification stops and we report the time for non-compliance of the VNFFG configuration as the time for non-compliance of the later consistency property (Figures 9 (right) and 11 (middle)). Since the hierarchy of the NFV stack implies that any faulty configuration at the higher levels would lead to a fault at the lower levels, to reduce the verification time, we exploit this observation and stop the verification once we have a violation at higher levels. Otherwise, we continue the process to verify the non-compliance of the consistency property between lower levels (e.g., L2/L3). In this case, the verification time is the time for verifying the consistency property between L1/L2 in case of no breach and the time for reporting non-compliance at the lower levels L2/L3 (Figures 9 (middle) and 11 (left)).

Scenario 1. Cross-Level Security Verification in Case of Compliance: Figure 9 (left) depicts the verification time for the consistency properties in case of compliance. In general, the consistency property verification consumes more time for verifying between higher levels (it requires 1∼5s for L1/L2) than for lower levels (1∼3s for L2/L3) due to a more complex and higher number of relation instances of the predicates between higher levels. Moreover, we also observe that with an increased number of VNFFGs, the required time is increasing almost linearly.

Scenario 2. Cross-Level Security Verification in Case of Detecting the First Breach: Figure 9 (middle and right) depict the verification time for the consistency properties in case of non-compliance and providing the evidence for the first security breach. The time to detect and report the first breach (∼ 3s while the first breach was found at L1/L2, and ∼ 6s

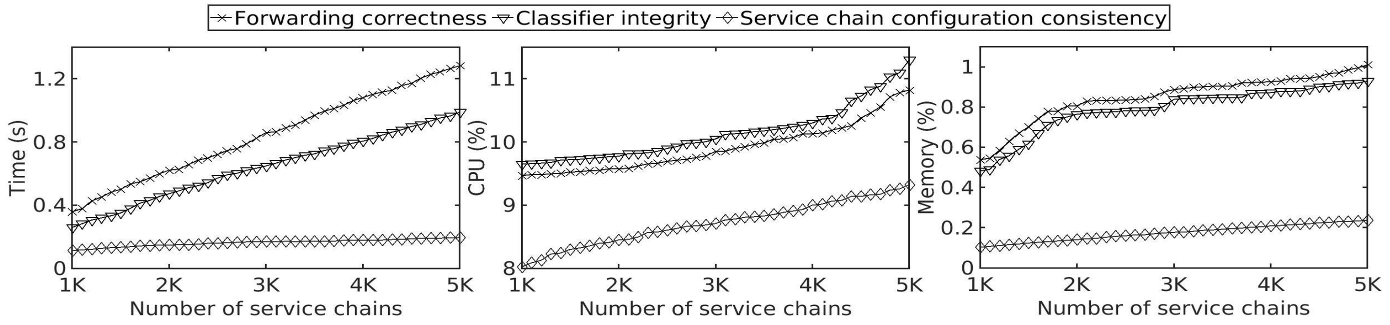


Figure 8: Verification performance for the consistency properties while varying the number of service chains.

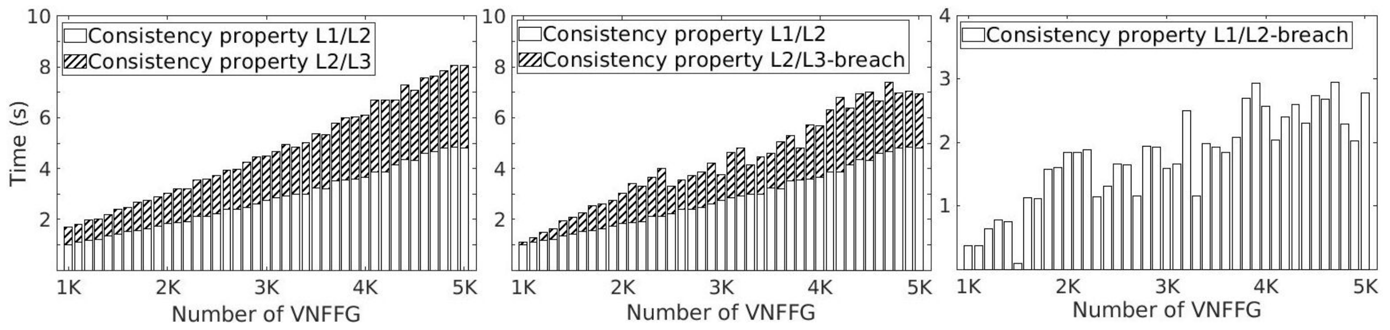


Figure 9: Verification time for the topology consistency properties in case of compliance (left), in case of reporting the first breach verifying between levels L2/L3 (middle), and in case of reporting the first breach verifying between L1/L2 (right).

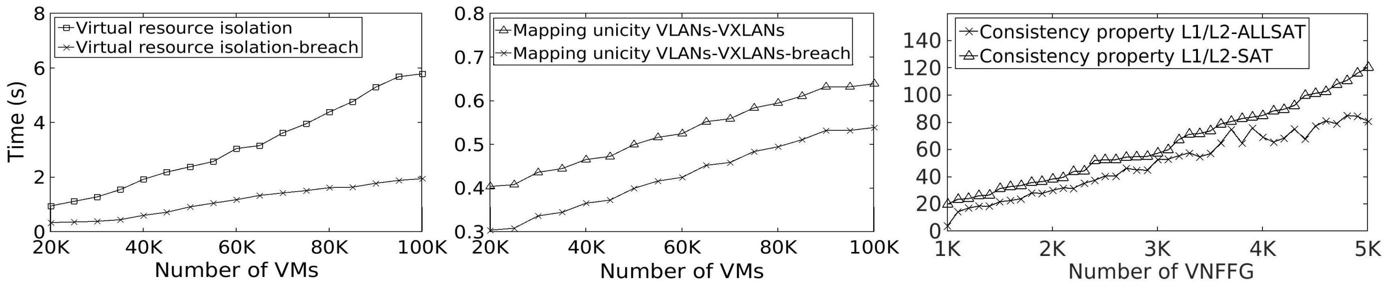


Figure 10: Verification time for the security properties virtual resource isolation (left) and mapping unicity VLANs-VXLANs (middle) in case of compliance and in case of reporting the first breach. Verification time for finding all compliance breaches (10 breaches) for the consistency property L1/L2 using SAT and ALLSAT solvers (right).

for L2/L3) is less than the time required for assessing the same property in case of compliance ($\sim$ 8s). This is due to the action of immediately stopping the verification process after finding the first breach as we mentioned earlier. Also, the time for detecting breaches between lower levels is not far from the time in the case of compliance, which can be attributed to the fact that the verification of consistency property between higher levels is more time-consuming than the one between lower levels. We consider the time for detecting non-compliance to be reasonable for application in real life as a non-real-time auditing solution.

Figure 10 (left and middle) show the time for verifying the security properties in case of compliance and reporting the evidence for the first breach. The verification of mapping unicity VLAN-VXLAN is more efficient (less than 1 second), and the required time increases more slowly than it does for the virtual resource isolation (6 seconds for the largest dataset) as the latter has more complex predicates involving a higher number of relation instances. Similarly, as in the case of consistency properties, the time for reporting the breach is shorter than the time for asserting compliance for both of the security properties.

Scenario 3. Cross-Level Security Verification in Case of Detecting All the Breaches: Figure 10 (right) shows the average verification time to find all compliance breaches for the consistency property L1/L2 for both the case of using SAT [22] and ALLSAT [52] solvers, while the given number of breaches in each dataset is 10. The figure shows that the ALLSAT solver is faster than SAT solver in finding all the security property breaches. The reason is that SAT solvers can only provide a single solution in each run, while ALLSAT solvers are capable of finding multiple breaches in a single run. As a ramification, to find all the solutions, we have to run the SAT solver again and again until finding all breaches (i.e., for determining 10 breaches in the experiment, we have to run the solver 10 times). Hence, ALLSAT is clearly more applicable when finding an exhaustive list of breaches is desirable.

Consequently, we analyze the efficiency of the ALLSAT solver in detecting all 10 breaches (Figure 11). Figure 11

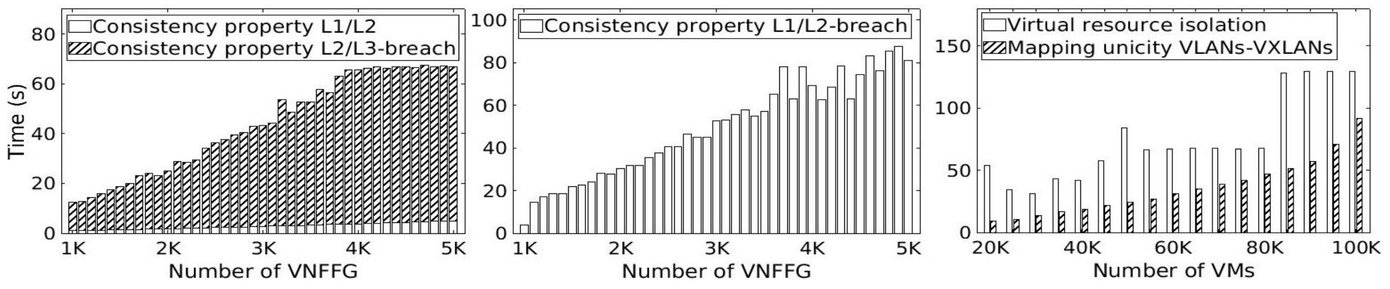


Figure 11: Verification time for the topology consistency properties, virtual resource isolation, and mapping unicity VLANs-VXLANs in case of reporting all compliance breaches using ALLSAT solver, with (left) reporting all breaches for verifying between levels L2/L3, (middle) reporting all breaches for verifying between L1/L2, and (right) reporting all breaches for verifying virtual resource isolation and mapping unicity VLANs-VXLANs.

(left) shows the average time for detecting non-compliance breaches at the lower levels (∼ 66s) for the largest dataset (5K VNFFGs), and Figure 11 (middle) shows the verification time for detecting non-compliance breaches at the higher levels (∼ 80s) for the same dataset. The ALLSAT solver could efficiently find all the violations, and the time for detecting multiple violations is longer than the time of detecting a single solution and assessing the compliance of the system in case of no breach. This indicates that the verification time of our solution increases with an increasing number of violations. Also, the time for finding all the breaches for the consistency property at L1/L2 is more than that for the consistency property between L2/L3, which is related to the complexity of the property at the higher levels. Therefore, the verification time for the lower levels is less than that of the higher levels, especially that the time for compliance verification of the consistency property at L1/L2 is as short as ∼ 5s. Moreover, even though the verification of the higher levels takes more time, its still much faster than the naive approach of verifying both of the properties, which would take both the time for verifying the consistency property between L1/L2 in case of non-compliance (i.e., ∼ 80s) and the time for verifying the consistency property between L2/L3 in case of non-compliance (i.e., ∼ 62s).

Figure 11 (right) shows the time for verifying the security properties in case of reporting all compliance breaches. The time of reporting all compliance breaches for both of the security properties is longer than the time for reporting compliance. Moreover, the time for reporting all breaches of the *virtual resource isolation* property (∼ 2m for the largest dataset (100K VMs)) is higher than the time for reporting all breaches of the *mapping unicity VLANs-VXLANs* property, as the latter is more complex. Figure 12 (left) studies the effect of increasing the number of violations on the verification time. In this experiment, we verify the *mapping unicity VLANs-VXLANs* property, and we vary the number of violations encountered in the dataset where the dataset size is (100K VMs). As depicted in the figure, the time increases almost linearly with the number of violations, and it takes about 3.7m to verify 50 breaches.

Efficiency Improvement Due to the ER Model. This set of experiments is to evaluate the efficiency improvement (Figure 13) resulting from utilizing the ER model in multi-level security verification by comparing its required time

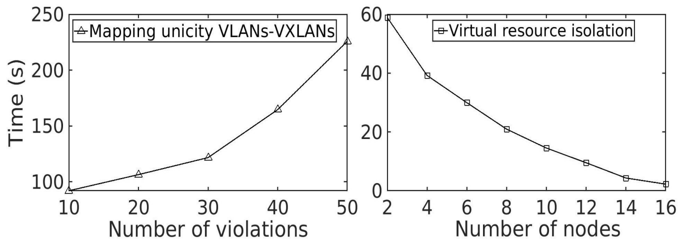


Figure 12: Verification time for reporting all breaches for the security property mapping unicity VLANs-VXLANs while varying the number of breaches (left) and the time for parallelizing the verification of the virtual resource isolation property (right).

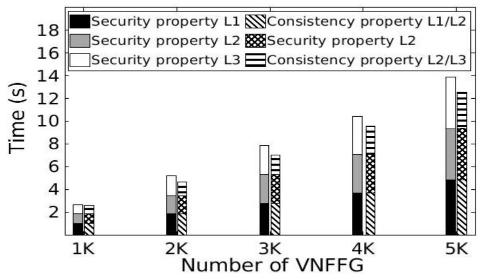


Figure 13: Comparing the verification time of the multi-level security property without (the grayscale bar) and with (the bar with patterns) the utilization of ER model.

with that of a conventional security verification approach (i.e., conducting security verification at each level). We verify the "SFC ordering and sequencing as defined by the specification" security property (defined in [28]), which checks if the deployed SFCs maintain the order of VNFs with the correct traffic forwarding behavior as defined by the specifications.

Figure 13 shows the required time for the multi-level verification for the "*SFC ordering and sequencing as defined by the specification*" security property. The grayscale bar represents verifying this property at each level of the NFV stack as mentioned in the motivating example (Section 1). The bars with patterns show the required time for verifying the same property with the existence of the ER model i.e., by verifying the consistency between the NFV stack levels after verifying the security property at one level (i.e., L2 in the figure, the middlebox with solid gray color). Each bar in the figure consists of three portions, where each portion represents the required time for verifying the security properties at each level or the consistency properties. With the help of the ER model, its possible to only conduct the security verification at one level (e.g., L2) and then conduct the consistency verification for the adjacent levels. Figure 13 depicts that the implementation of the ER model reduces the verification time; for instance, for the largest dataset (5K SFCs), the implementation of the ER model reduces the overall verification time by 1.4 seconds.

Applicability of NFVGuard+ to Different Solvers. The intention of this experiment is to investigate the applicability of NFVGuard+ to different SAT solvers. Our implementation is based on Sugar, which is an SAT-based constraint solver, where the CSP is solved by a backend SAT solver. Sugar supports MiniSat [22] as the default backend SAT solver. Our next experiment also investigates ALLSAT (i.e., short for all solutions SAT) backend solver [53], a variant of SAT solvers that deals with enumerating all satisfying assignments of a propositional logic formula. To the best of our knowledge, clasp [54], PicoSAT [52], and relsat [55] are the only ALLSAT solvers. Since PicoSAT is the only ALLSAT solver supported by Sugar, we consider this in our implementation.

More specifically, to demonstrate the applicability of NFVGuard+ to different solvers, we implement Sugar to assess the consistency properties at L1/L2 using ALLSAT (i.e., PicoSAT) and SAT (i.e., MiniSat) solvers. Figure 14 illustrates this verification performance in terms of time, CPU, and memory. Figure 14 depicts that the performance of both ALLSAT and SAT solvers is mostly similar. Generally, for both these solvers, resource consumption increases almost linearly with the increased number of VNFFGs. The ALLSAT solver requires a slightly longer verification time (Figure 14 (left)); to be specific, ALLSAT takes $\sim$ 1.3 seconds more than the SAT solver to verify the same property for the largest dataset (5K VNFFGs). On the other hand, ALLSAT solver consumes less CPU, while the memory consumption is almost the same for both solvers. On the other hand, though ALLSAT solvers are slower than SAT solvers, the required time by ALLSAT solver to identify multiple breaches (especially for a larger number of breaches) is less than an SAT solver as we described earlier in Figure 10 (right). Hence, we can conclude that NFVGuard+ is not solver dependent, and hence a user should choose the solver based on his/her requirements (e.g., find multiple breaches at a time or one by one).

Parallel Execution of the Properties. We can reduce the required time by verifying the properties in a parallel manner. Though different approaches are used to parallel verification [56], Sugar unfortunately, does not support parallelization. Hence, we adopt the search space splitting technique [56] which adopts a similar logic as in other parallel verification approaches. Specifically, in our technique, we split the audit data across multiple CSP instances that implement the same property rather than splitting the search space. In this way, we reduce the payload of verifying one large CSP instance by verifying less volume of audit data, and we can run the CSP instances in parallel. We choose the virtual resource isolation property because its the most resource-consuming property in case of detecting all non-compliance breaches (refer to Figure 11 (right)), and we also evaluate using the largest dataset with 100K VMs. As shown in Figure 12 (right), the time for the first round (two CSP instances) is reduced by 55% and the required time continues to decrease until we reach a reduction of 99%.

7.2 Experiments with Real Data

We apply NFVGuard+ to the real data collected from a real infrastructure hosted at one of the largest telecommunications vendors. The examined part of the infrastructure is composed of two racks, connected to two edge switches, which are connected to two aggregate switches, as depicted in Figure 15. The data contains 20 tenants, 111 VMS, 9 subnets, 26 physical servers, 26 vSwitches, 679 OvS flows, 35 VLANs, and 9 VXLANs. We apply NFVGuard+ to verify this real data against various properties. We report the average findings among those properties in Table 4. The resource consumption in terms of time, CPU, and memory increases with the amount of data as shown in Table 4. This result also follows a similar trend to what we found for the synthetic data in previous experiments. We can also observe that the values of resource consumption in this experiment are generally much smaller than in previous experiments performed using synthetic datasets (which were deliberately scaled up to evaluate the scalability of our solution).

Performance metrics	Percentage of dataset				
	20%	40%	60%	80%	100%
Time (S)	0.78	0.84	0.88	0.90	0.93
CPU (%)	2.48	2.57	2.62	2.65	2.66
Memory (%)	0.041	0.044	0.046	0.046	0.047

Table 4: The experimental results of NFVGuard+ for the real data. The average time, CPU, and memory required for the verification of three sample NFV security properties, i.e., VNFs co-residence, virtual resource isolation, and mapping unicity VLANs-VXLANs, based on real data.

8 Discussion

A Guideline to Adapt NFVGuard+ to Other NFV Platforms. NFVGuard+ utilizes the constructed ER model to identify the audit data and formulate the NFV security properties. Although the ER model is based on OpenStack-/Tacker, its general enough to be extended to other platforms, especially because we capture high-level components related to the general concept of NFV that are common to most of the deployments. We detail how the ER model will change at each level if we consider different implementation platforms as follows.

The first level in the ER model represents the entities created at the service orchestration level after processing the network service design specification (NSD) from the NFV user/provider. At this level, platforms such as ONAP [30], OSM [57], and Tacker are employed to enable the design, creation, orchestration, and auto-scaling of services on top of the resource management and virtual infrastructure layers. In our model, we depict high-level components related to the network service itself (not on the deployed platform), therefore, our model at this level is general and can be extended to different deployments. However, the scripts to collect and correlate data may vary with different deployments, especially when the data needs to be extracted from the configuration files specific to the deployed platform.

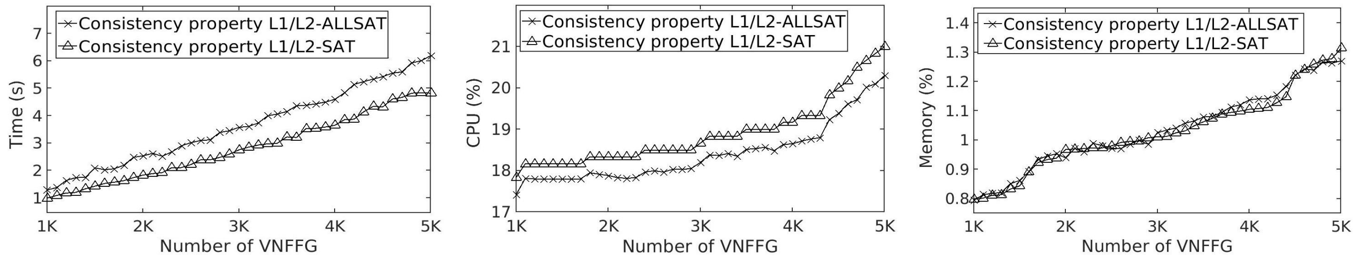


Figure 14: Verification performance for the consistency property L1/L2 using ALLSAT and SAT solvers.

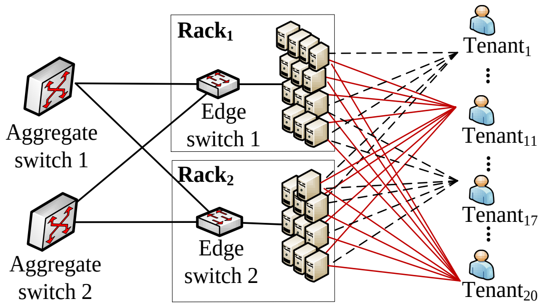


Figure 15: The topology of a part of a real cloud data center operating NFV used in our experiments.

The same thing follows at the second and third levels of our ER model. Different cloud platforms can be deployed at L2 to instantiate the network services, and most of them offer similar capabilities of creating, provisioning, and managing the virtual resources required for instantiating the network services. Our model captures the main virtual components that are common to those platforms.

For instance, we consider a specific virtual infrastructure level implementation mainly relying on VLAN and VXLAN as well-established network virtualization technologies and OvS as a widely used virtual switch implementation. Other platforms may support different virtualization technologies such as Generic Routing Encapsulation (GRE) [58] or Generic Network Virtualization Encapsulation (GENEVE) [59]. In this case, the entities of the ER model at this level will change but not significantly (e.g., replacing the VXLAN entity with GRE), and the properties may either remain applicable or need to be modified or skipped. As an example, in the case of small to medium clouds, where VLAN tags are sufficient to implement all L3 virtual networks on top of the physical network, the ER model will be simplified, and the security properties related to the mapping between VLAN and VXLAN become unnecessary.

In summary, our ER model and properties formulation cover high-level virtualization components that are common to most deployments. Therefore, it can be adapted to most of the deployments with minor changes. The scripts to collect and process audit data need to be revised according to the implementation details of each deployment. However, this is a one-time effort that is only needed before initializing the verification process.

Scope of the Security Verification. The security verification is conducted at a specific level(s) based on the definition of the verified security property. For example, the "Mapping unicity VLANs-VXLANs" security property (defined in [28]) can be verified only at L3 of the NFV stack because the data relevant to it exists at that level. Other properties, such as the "SFC ordering and sequencing as defined by the specification" security property (defined in [28]) require collecting data from various levels such as L1, L2, or L3. In this case, it depends on the auditor to define the specifications of the property. Therefore, the verification is property-dependent and sometimes can extend to all levels to ensure the correctness of the security property in the entire stack.

Automated Implementation. Since NFVGuard+ works with a static snapshot of the NFV environment, to maintain the security of the audited system, it needs to run periodically or on-demand when a change is made to the system. To that end, setting the period between verifications could be critical: a large interval between two verifications could lead to undetected security breaches and a small interval might incur prohibitive overhead. Hence we intend to improve the efficiency of our approach by exploring and adopting incremental [60] or proactive [61] techniques. Moreover, our current approach requires some manual effort and expertise in constructing the ER model, identifying the security properties, and formally encoding them. Although most of these efforts is done only once, we aim to automate those processes in our future work.

Limitations. NFVGuard+ focuses on verifying the compliance of the NFV stack with respect to consistency properties and security properties. Specifically, the properties within the scope of this paper include those pertaining to the static configuration of the virtualized infrastructure. This involves ensuring the proper configuration of isolation mechanisms and maintaining topology consistency. Out of scope properties include dynamic properties, such as those related to reachability and network forwarding functionality. Although these properties can be verified using formal methods, they will be addressed in future work. Furthermore, while our approach can detect violations of security and consistency properties that may result from vulnerability exploitations, threats, or attacks, its not designed to attribute such a violation to specific underlying vulnerabilities (i.e., vulnerability analysis) or particular attacks (i.e., intrusion detection). Additionally, it does not detect violations that are not reflected in logs and configurations, as the accuracy of our audit results relies on the input data extracted from these sources.

9 Related Work

Most existing security verification solutions (e.g., [4]–[18], [24]) in NFV focus on the verification of one particular level (mostly SFC). In particular, ChainGuard [11], SFC-Checker [13], Cohen et al. [15], and AuditBox [16], all verify the correct forwarding behavior of SFCs. Other solutions, including NFVSense [6], CloudVaults [7], APPD [18], and Cheng et al.

Solution	NFV	Levels			Property	Method
		L1	L2	L3		
[17]	✓	✓	✗	✗	Security	Graph theoretic
[35]	✓	✗	✓	✗	Network	Custom algorithms
[10]–[12]	✓	✗	✗	✓	Correctness, performance	Trusted shim layer, graph theoretic
[4]–[9], [13]–[16], [18], [24]	✓	✗	✗	✗	Network, correctness, integrity	Remote attestation, MaxSAT solver, graph theoretic, custom algorithms, trusted shim, verified routing protocol, packet pair dispersion, tag-based verification, and machine learning
[36], [62]–[65]	✗	✗	✗	✗	Security, operational, network, identity and access control	Graph theoretic, CSP solver, custom algorithms
NFVGuard+, [25]	✓	✓	✓	✓	Security and consistency	CSP solver, machine learning

Table 5: Summary of existing solutions. The symbols (✓) and (✗) mean supported and not supported, respectively.

[24], focus on SFC integrity verification. vSFC [8] verifies various SFC violations (e.g., packet injection attacks and path non-compliance) and vHSFC [5] utilizes a lightweight Verified Routing Protocol (VRP) to detect various hybrid SFC violations and attacks. EnsureS [4] introduces an SFC path validation model that employs batch hashing and tag verification. VeriNeS [17] proposes a runtime verification framework for detecting anomalies in network services. In contrast, Zoure et al. [66] investigate NFV network service anomalies and the challenges in achieving verification.

Several solutions (e.g., [9], [10], [12], [14]) focus on verifying SFCs functionality and performance. They cover a wide range of verification aspects, such as performance and accounting [10], SLA-related performance properties [12], verification of reachability policies [14], and detection of dependencies and conflicts between network functions [9]. Unlike all those works, the main focus of our approach is to ensure the security of an NFV stack at all levels. Also, unlike us, most of those works do not formally model the verification problem.

There are a few solutions (e.g., [25], [26]) that tackle the multi-level aspect of NFV. Lakshmanan et al. [25] propose employing Neural Machine Translation (NMT) to detect cross-level inconsistency attacks. However, their utilization of NMT for detection is considered less reliable in terms of accuracy compared to FMs. On the other hand, Alhebaishi et al. [26] model and address cross-layer and co-residency attacks through VM placement optimization, focusing on a narrower range of attacks compared to our approach.

Also, there exist other works (e.g., [36], [37], [41], [62]–[65], [67]) that verify security properties in virtual networks, e.g., clouds and SDN. Among them, ISOTOP [41] and Xu et al. [68] cover the consistency between different cloud layers. Additionally, there are other solutions, e.g., Net-Plumber [63], Veriflow [64], and NoD [37] 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. Table 5 compares existing solutions with NFVGuard+. It lists the solutions, whether they target NFV or other virtual environments, the NFV stack level they address, and the verified properties along with their verification methods.

10 Conclusion

We presented NFVGuard+, a novel approach to the formal cross-level security verification of the NFV stack. Specifically, we proposed a system entity-relationship (ER) model that captures the detailed mappings and the relationships between the NFV resources across different levels in the NFV stack and devised a system that offers an assisted solution for NFV users to identify and verify the NFV properties by leveraging the ER mode. We implemented a real NFV testbed using OpenStack/Tacker, integrated our solution into the testbed, and evaluated our approach through experiments using synthetic data and real data provided by one of the largest telecommunications vendors. The results confirmed the efficiency and real-life applicability of our approach. In future work, we plan to explore more efficient techniques using incremental, or proactive verification for further improvement in terms of efficiency and scalability. Additionally, although the general approach of NFVGuard+ is platform-agnostic, the current implementation of data collection and processing is still limited to OpenStack/Tacker. Therefore, we will address this limitation in our future work through a more modular design with a concrete methodology for extending to other open-source NFV platforms (e.g., OPNFV and OSM).

References

[1] Omdia, "NFV/Edge Adoption and Vendor Perception Survey," 2021, available at: https://omdia.tech.informa.com/OM019961/NFVEdge-Adoption-and-Vendor-Perception-Survey–2021.

[2] 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.

[3] 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.

[4] S. Pradeep, Y. K. Sharma, U. K. Lilhore, S. Simaiya, A. Kumar, S. Ahuja, M. Margala, P. Chakrabarti, and T. Chakrabarti, "Developing an sdn security model (ensures) based on lightweight service path validation with batch hashing and tag verification," *Scientific Reports*, vol. 13, no. 1, p. 17381, 2023.

[5] S. Chen, J. Li, B. Chen, D. Guo, and K. Li, "vhsfc: Generic and agile verification of service function chain with parallel vnfs," in *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2023, pp. 498–503.

[6] M. Oqaily, S. Majumdar, L. Wang, M. Ekramul Kabir, Y. Jarraya, A. Asadujjaman, M. Pourzandi, and M. Debbabi, "A tenant-based two-stage approach to auditing the integrity of virtual network function chains hosted on third-party clouds," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, 2023, pp. 79–90.

[7] B. Larsen, H. B. Debes, and T. Giannetsos, "Cloudvaults: Integrating trust extensions into system integrity verification for cloud-based environments," in *Computer Security: ESORICS 2020 International Workshops, DETIPS, DeSECSys, MPS, and SPOSE, Guildford, UK, September 17–18, 2020, Revised Selected Papers 25*. Springer, 2020, pp. 197–220.

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

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

[10] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar, "Verifiable network function outsourcing: Requirements, challenges, and roadmap," in *HotMiddlebox*, 2013, pp. 25–30.

[11] M. Flittner, J. M. Scheuermann, and R. Bauer, "Chainguard: Controller-independent verification of service function chaining in cloud computing," in *IEEE (NFV-SDN)*, 2017, pp. 1–7.

[12] 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.

[13] 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.

[14] G. Marchetto, R. Sisto, F. Valenza, J. Yusupov, and A. Ksentini, "A formal approach to verify connectivity and optimize vnf placement in industrial networks," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 2, pp. 1515–1525, 2020.

[15] R. Cohen, L. Katzir, and A. Yehezkel, "Efficient service chain verification using sketches and small samples," in *2021 IEEE (NFV-SDN)*. IEEE, 2021, pp. 1–7.

[16] G. Liu, H. Sadok, A. Kohlbrenner, B. Parno, V. Sekar, and J. Sherry, "Don't yank my chain: Auditable NF service chaining," in *18th USENIX (NSDI'21)*, 2021, pp. 155–173.

[17] M. Zoure, T. Ahmed, and L. Réveillère, "VeriNeS: Runtime verification of outsourced network services orchestration," in *36th ACM*, 2021, pp. 1138–1146.

[18] A. Asadujjaman, M. Oqaily, Y. Jarraya, S. Majumdar, M. Pourzandi, L. Wang, and M. Debbabi, "Artificial Packet-Pair Dispersion (APPD): A Blackbox Approach to Verifying the Integrity of NFV Service Chains," in *2021 IEEE (CNS)*. IEEE, 2021, pp. 245–253.

[19] National Institute of Standards and Technology, "CVE-2024-1085 Detail," 2024, https://nvd.nist.gov/vuln/detail/CVE-2024-1085. Last accessed 19 May 2024.

[20] National Institute of Standards and Technology, "CVE-2024-0193 Detail," 2024, https://nvd.nist.gov/vuln/detail/CVE-2024-0193. Last accessed 19 May 2024.

[21] National Institute of Standards and Technology, "CVE-2024-0646Detail," 2024, https://nvd.nist.gov/vuln/detail/CVE-2024-0646. Last accessed 19 May 2024.

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

[23] OpenStack, "OpenStack," 2020, available at: https://www.openstack.org/.

[24] S.-T. Cheng, C.-Y. Zhu, C.-W. Hsu, and J.-S. Shih, "The anomaly detection mechanism using extreme learning machine for service function chaining," in *2020 International Computer Symposium (ICS)*. IEEE, 2020, pp. 310–315.

[25] S. Lakshmanan, M. Zhang, S. Majumdar, Y. Jarraya, M. Pourzandi, and L. Wang, "Caught-in-translation (cit): Detecting cross-level inconsistency attacks in network functions virtualization (nfv)," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[26] N. Alhebaishi, L. Wang, and S. Jajodia, "Modeling and mitigating security threats in network functions virtualization (nfv)," in *Data and Applications Security and Privacy XXXIV: 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25–26, 2020, Proceedings 34*. Springer, 2020, pp. 3–23.

[27] OpenStack, "Verizon launches industry-leading large OpenStack NFV deployment," 2016, available at: https://www.openstack.org/news/view/215/verizon-launches-industry-leading-large-openstack-nfv-deployment.

[28] A. Oqaily, L. Sudershan, Y. Jarraya, S. Majumdar, M. Zhang, M. Pourzandi, L. Wang, and M. Debbabi, "NFVGuard: Verifying the Security of Multilevel Network Functions Virtualization (NFV) Stack," in *2020 IEEE (CloudCom)*. IEEE, 2020, pp. 33–40.

[29] "ETSI: Network Functions Virtualisation Architectural Framework," https://www.etsi.org/. Last accessed 16 June 2022.

[30] ONAP, "Open Network Automation Platform," 2022, available at: https://www.onap.org.

[31] OpenStack, "OpenStack Tacker," 2020, https://wiki.openstack.org/wiki/Tacker. Last accessed 16 June 2022.

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

[33] IETF, SFC, "Internet Engineering Task, SFC Active WG Working Group Documents," 2020. [Online]. Available: https://www.redhat.com/en/blog/2018-year-open-source-networking-csps

[34] "Cloud Security Alliance," available at: https://cloudsecurityalliance.org/research/ccm/.

[35] G. S. Chawla, M. Zhang, S. Majumdar, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "VMGuard: State-based proactive verification of virtual network isolation with application to NFV," *IEEE (TDSC)*, vol. 18, no. 4, pp. 1553–1567, 2020.

[36] 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.

[37] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *12th USENIX NSDI*, 2015.

[38] F. Sierra-Arriaga, R. Branco, and B. Lee, "Security issues and challenges for virtualization technologies," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–37, 2020.

[39] D. Tank, A. Aggarwal, and N. Chaubey, "Virtualization vulnerabilities, security issues, and solutions: a critical study and comparison," *International Journal of Information Technology*, pp. 1–16, 2019.

[40] Linux Foundation, "Open vSwitch," 2016.

[41] 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*, vol. 22, no. 1, pp. 1:1–1:35, 2018.

[42] 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.

[43] A. Wang, M. Iyer, R. Dutta, G. N. Rouskas, and I. Baldine, "Network virtualization: Technologies, perspectives, and frontiers," *Journal of Lightwave Technology*, vol. 31, no. 4, pp. 523–537, 2012.

[44] OpenStack, "Heavy reading study on CSPs and OpenStack," 2016, https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/pdf-downloads/OpenStack-survey-results-public-presentation.pdf. Last accessed 16 June 2022.

[45] Oasis, "Topology and Orchestration Specification for Cloud Applications (TOSCA)," 2013, available at:https://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf.

[46] H. Hawilo, M. Jammal, and A. Shami, "Exploring microservices as the architecture of choice for network function virtualization platforms," *IEEE Network*, vol. 33, no. 2, pp. 202–210, 2019.

[47] OpenStack, "OSSA-2017-004: OpenStack - Incorrect role assignment with federated keystone," 2017, available at: https://security.openstack.org/ossa/OSSA-2017-004.html.

[48] D. Ishii and S. Fujii, "Formalizing the soundness of the encoding methods of sat-based model checking," in *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2020, pp. 105–112.

[49] J. C. Blanchette, M. Fleury, P. Lammich, and C. Weidenbach, "A verified sat solver framework with learn, forget, restart, and incrementality," *Journal of automated reasoning*, vol. 61, pp. 333–365, 2018.

[50] S. Manandhar, K. Singh, and A. Nadkarni, "Towards automated regulation analysis for effective privacy compliance," in *ISOC Network and Distributed System Security Symposium*, 2024.

[51] M. W. P. Shuvo, M. N. Hoq, S. Majumdar, and P. Shirani, "On reducing underutilization of security standards by deriving actionable rules: An application to iot," in *International Conference on Research in Security Standardisation*. Springer, 2023, pp. 103–128.

[52] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, no. 2-4, pp. 75–97, 2008.

[53] T. Toda and T. Soh, "Implementing efficient all solutions SAT solvers," *JEA*, vol. 21, pp. 1–44, 2016.

[54] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "clasp: A conflict-driven answer set solver," in *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 2007, pp. 260–265.

[55] R. J. Bayardo Jr and J. D. Pehoushek, "Counting models using connected components," in *AAAI/IAAI*, 2000, pp. 157–162.

[56] R. Martins, V. Manquinho, and I. Lynce, "An overview of parallel SAT solving," *Constraints*, vol. 17, no. 3, pp. 304–347, 2012.

[57] OSM, "Open Source MANO," 2022, available at: https://osm.etsi.org/.

[58] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Rfc2784: Generic Routing Encapsulation (GRE)," 2000.

[59] J. Gross, I. Ganga, and T. Sridhar, "Rfc 8926 geneve: Generic network virtualization encapsulation," 2020.

[60] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "User-level runtime security auditing for the

cloud," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1185–1199, 2017.

[61] 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*. Springer, 2017, pp. 265–285.

[62] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim, "Proactive security analysis of changes in virtualized infrastructures," in *ACSAC*, 2015, pp. 51–60.

[63] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *10th USENIX NSDI'13*, 2013.

[64] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide invariants in real time," in *10th USENIX (NSDI'13)*, 2013, pp. 15–27.

[65] 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.

[66] M. Zoure, T. Ahmed, and L. Réveillère, "Network services anomalies in nfv: Survey, taxonomy, and verification methods," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1567–1584, 2022.

[67] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *19th USENIX (NSDI'12)*, 2012, pp. 113–126.

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

PLACE
PHOTO
HERE

Alaa Oqaily She is currently a Ph.D. student at Concordia University, Canada. She received her MSc degree from Jordan University of Science and Technology, Jordan. Her research interests include cloud computing security, NFV security, and issues related to machine learning.

Mohammad Ekramul Kabir He is a Horizon postdoctoral research fellow at Concordia University, Canada. He received his PhD in Information and Systems Engineering from Concordia University. His research interests include green, smart, and secure charging of electric vehicles, cloud/edge computing security, and applications of artificial intelligence.

Lingyu Wang received a PhD in information technology from George Mason University in the USA. He is a professor with the Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Quebec, Canada. He holds the NSERC/Ericsson Industrial Research Chair in SDN/NFV Security. His research interests include SDN/NFV security, cloud computing security, network security metrics, software security, and privacy.

Yosr Jarraya received a PhD from Concordia University in 2010 and has been a master researcher at Ericsson since 2016, focusing on security and privacy in cloud, SDN, and NFV. She holds several patents and has co-authored two books and over 40 research papers in prestigious journals and conferences such as the ACM Transactions on Privacy and Security, NDSS, ESORICS, etc.

Suryadipta Majumdar received the PhD from Concordia University, Canada, where he is currently a Gina Cody Research and Innovation Fellow and an associate professor. His research focuses on cloud security, SDN security, and IoT security.

Makan Pourzandi received the MSc in parallel computing from Ecole Normale Superieure de Lyon, France, and the PhD in computer science from the University of Lyon I, France. Currently, he is a researcher with Ericsson, Canada, with over 15 years of experience in telecom systems security, cloud computing, distributed systems security, and software security.

Mourad Debbabi received the MSc and PhD degrees in computer science from Paris-XI Orsay University, Orsay, France. He is a full professor at Concordia University, serving as dean of 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. With over 300 peer-reviewed publications, he has supervised 32 PhD students, 76 master's students, and 14 postdoctoral fellows, and has held prominent positions in academia and industry, including Panasonic and General Electric.

Sudershan L T He is currently a Jr. Information Security Analyst at SAP. He received his MSc in Information Systems Security from Concordia University, Montreal, Quebec, Canada, and his M.Sc. in Software Engineering from Coimbatore Institute of Technology, Coimbatore, India. Previously, he worked as a Software/Network Security Engineer at Zoho Corporation Pvt. Ltd., India. His research interests include cloud computing security and privacy, network security, and software security.

Mengyuan Zhang received her PhD in Information and Systems Engineering from Concordia University, Montreal, Canada. Following her doctoral studies, she worked as a researcher at Ericsson Research, Montreal. Currently, she is an assistant professor in the Department of Computer Science at Vrije Universiteit Amsterdam. Previously, she was a research assistant professor in the Department of Computing at the Hong Kong Polytechnic University. Her research interests include security metrics, attack surfaces, cloud computing security, and applied machine learning in security, with numerous publications in leading peer-reviewed journals and conferences.