

*Caught-in-Translation (CiT): Detecting Cross-level  
Inconsistency Attacks in Network Functions Virtualization*

Sudershan Lakshmanan Thirunavukkarasu

A Thesis  
in  
The Department  
of  
Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements  
for the Degree of  
Master of Applied Science (Information Systems Security) at  
Concordia University  
Montréal, Québec, Canada

August 2020

© Sudershan Lakshmanan Thirunavukkarasu, 2020

CONCORDIA UNIVERSITY  
School of Graduate Studies

This is to certify that the thesis prepared

By: **Sudershan Lakshmanan Thirunavukkarasu**  
Entitled: ***Caught-in-Translation (CiT): Detecting Cross-level Inconsistency***  
**Attacks in Network Functions Virtualization**

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

**Master of Applied Science (Information Systems Security)**

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

Signed by the final examining committee:

\_\_\_\_\_ Chair  
*Dr. Mohsen Ghafouri*

\_\_\_\_\_ External Examiner  
*Dr. Olga Ormandjieva*

\_\_\_\_\_ Internal Examiner  
*Dr. Chadi Assi*

\_\_\_\_\_ Supervisor  
*Dr. Lingyu Wang*

\_\_\_\_\_ Co-supervisor  
*Dr. Mengyuan Zhang*

Approved by \_\_\_\_\_  
Chair of Department or Graduate Program Director

August, 2020

\_\_\_\_\_  
Dr. Mourad Debbabi, Dean  
Gina Cody School of Engineering and Computer Science

# Abstract

## *Caught-in-Translation (CiT): Detecting Cross-level Inconsistency Attacks in Network Functions Virtualization*

Sudershan Lakshmanan Thirunavukkarasu

By providing network functions through software running on standard hardware, Network Functions Virtualization (NFV) brings many benefits, such as increased agility and flexibility with reduced costs, as well as additional security concerns. Although existing works have examined various security issues of NFV, such as vulnerabilities in VNF software and DoS, there has been little effort on a security issue that is intrinsic to NFV, i.e., as an NFV environment typically involves multiple abstraction levels, the inconsistency that may arise between different levels can potentially be exploited for security attacks. Existing solutions mostly focus on verification, which is after the fact and cannot prevent irreversible damages. Further adding to the complexity, the different abstraction levels can be managed by multiple service providers, which may render the data required for verification inaccessible. Moreover, many existing solutions are limited to a single abstraction level and disregard the multi-level nature of NFV.

In this work, we propose the first NFV deployment model to capture the deployment aspects of NFV at different abstraction levels, which is essential for an in-depth study of the inconsistencies between such levels. We then present concrete attack scenarios in which the inconsistencies are exploited to attack the network functions in a stealthy manner. Based on the deployment model, we study the feasibility of detecting the inconsistencies through verification. Furthermore, by drawing an analogy between multi-level NFV events and natural languages, we propose a Neural Machine Translation (NMT)-based detection

approach, namely, *Caught-in-Translation (CiT)*, to detect cross-level inconsistency attacks in NFV. Specifically, we first extract event sequences from different abstraction levels of an NFV stack. We then leverage the Long Short-Term Memory (LSTM) to translate the event sequences from one level to another. Finally, we apply both similarity metric and Siamese neural network to compare the translated event sequences with the actual sequences to detect attacks. We integrate *CiT* into OpenStack/Tacker, and evaluate its performance using both real and synthetic data. Experimental results show that *CiT* outperforms traditional anomaly detection and provides an accurate, efficient, and robust solution for detecting inconsistency attacks in NFV.

# Acknowledgments

Firstly, I would like to thank my thesis advisor, Dr. Lingyu Wang. His continuous support, guidance and belief in my research abilities helped me the most to finish this thesis work successfully. Whenever I reached out to him with issues related to my work, he always put me on the right path. With his inspiring speeches, he kept me inspired, motivated and driven towards achieving the results. I am extremely fortunate to have had him as my supervisor. I would also like to extend my thanks to my co-supervisor, Dr. Mengyuan Zhang, for the time and effort she has invested to help me with my research. She has always made me think out of the box by asking the right questions. Her invaluable support and guidance helped me throughout my research and for completing my master's thesis.

I would also like to express my deep gratitude to the members of Audit Cloud Ready (ARC) research group from both Concordia University and Ericsson Research Canada for all the insightful and enthusiastic discussions that has helped me in several occasions. I thank all other faculty members of the CIISE department. I also thank all the examining committee members of my master thesis defence.

Finally, I would like to acknowledge the unconditional support given by my family and friends throughout the years of my master's study. This accomplishment would not have been possible without them.

# Contents

<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Inconsistencies in an NFV Stack . . . . .	1
1.2 Thesis Statement . . . . .	3
1.3 Contributions . . . . .	4
1.4 Thesis Organization . . . . .	5
<b>2 NFV Deployment Model</b>	<b>6</b>
2.1 Preliminaries . . . . .	6
2.1.1 The ETSI NFV Reference Architecture . . . . .	6
2.1.2 NFV Service Models . . . . .	9
2.1.3 The Cloud Architecture . . . . .	9
2.2 Motivating Example . . . . .	10
2.3 The NFV Deployment Model . . . . .	12
2.3.1 Overview . . . . .	12
2.3.2 L1: Service Orchestration Level . . . . .	14
2.3.3 L2: Resource Management Level . . . . .	15

2.3.4	L3: Virtual Infrastructure Level . . . . .	16
2.3.5	L4: Physical Infrastructure Level . . . . .	17
<b>3</b>	<b>Inconsistencies in NFV Stack</b>	<b>18</b>
3.1	Overview . . . . .	18
3.2	Threat Model . . . . .	20
3.3	Attacks targeting the consistency property . . . . .	20
3.3.1	Attacks at L2 . . . . .	20
3.3.2	Attacks at L3 . . . . .	21
3.4	Attack Scenarios . . . . .	21
3.4.1	Attack Scenario 1 . . . . .	22
3.4.2	Attack Scenario 2 . . . . .	24
3.5	Feasibility of Consistency Verification . . . . .	25
<b>4</b>	<b>Detecting Cross-level Inconsistency Attacks in NFV</b>	<b>27</b>
4.1	Motivating Example . . . . .	27
4.2	Key Ideas . . . . .	29
4.3	CiT Overview . . . . .	30
4.3.1	Training Phase . . . . .	31
4.3.2	Detection Phase . . . . .	31
4.4	Data Preparation and Event Embedding Generation . . . . .	32
4.4.1	Data Preparation . . . . .	32
4.4.2	Event Embedding Generation . . . . .	35
4.5	Neural Machine Translation . . . . .	36
4.5.1	Background on LSTM . . . . .	36
4.5.2	Event Sequence Translation . . . . .	36
4.6	Neural Network-based Inconsistency Detection . . . . .	38

4.6.1	Background . . . . .	38
4.6.2	Training . . . . .	39
4.6.3	Inconsistency Detection . . . . .	40
4.6.4	Diff-based TOSCA Verifier . . . . .	42
<b>5</b>	<b>Implementation and Experiments</b>	<b>44</b>
5.1	Implementation of NFV Testbed . . . . .	44
5.1.1	Onboarding Network Service Descriptors . . . . .	46
5.1.2	Instantiating the Network Service . . . . .	46
5.1.3	NFV Testbed Components . . . . .	47
5.2	Experiments . . . . .	52
5.2.1	Implementation Details and Experimental Settings . . . . .	52
5.2.2	Datasets . . . . .	54
5.2.3	Challenges in Processing the Real-World Data . . . . .	55
5.2.4	Evaluation on Out-of-Vocabulary (OOV) Events . . . . .	56
5.2.5	Event Embedding Model . . . . .	58
5.2.6	Inconsistency Detection Evaluation . . . . .	59
5.2.7	Robustness Evaluation . . . . .	67
5.2.8	Hyperparameter Selection . . . . .	70
5.2.9	Efficiency and Scalability . . . . .	76
<b>6</b>	<b>Discussion</b>	<b>81</b>
6.1	Additional Use Cases . . . . .	81
6.2	Feasibility of Training . . . . .	82
6.3	Employing Attention-based Translation Mechanisms . . . . .	82
6.4	Adapting to other NFV Platforms . . . . .	83
6.5	Limitations . . . . .	83



<b>7</b>	<b>Related Work</b>	<b>84</b>
7.1	NFV Models . . . . .	84
7.2	NFV Security . . . . .	84
7.3	Anomaly Detection on Sequential Data . . . . .	85
7.4	Translation-based Security Approaches . . . . .	86
<b>8</b>	<b>Other Contributions</b>	<b>87</b>
8.1	NFVGuard: Verifying the Security of Multilevel Network Functions Virtu- alization (NFV) Stack . . . . .	87
8.2	NFV Testbed Deployment . . . . .	88
8.2.1	Implementation Challenges . . . . .	88
<b>9</b>	<b>Future Work and Conclusion</b>	<b>91</b>
9.1	Future Work . . . . .	91
9.2	Conclusion . . . . .	92
	<b>Bibliography</b>	<b>93</b>

# List of Figures

1	The ETSI NFV reference architecture [19] . . . . .	8
2	The NIST cloud model [59] depicting the two-layer view of a multi-tenant virtualized infrastructure in cloud . . . . .	10
3	An inconsistency between the NS specification and instance . . . . .	11
4	The multilevel NFV deployment model . . . . .	13
5	An implemented attack at L2 causing inconsistency between the path specification <code>VNFFG1:NFP1</code> and its instance <code>nfp1-chain</code> . . . . .	22
6	An illustration of the attack timeline when Alice modifies Bob's port-chain <code>nfp1-chain</code> by adding a malicious <code>mVDU</code> at L2 . . . . .	23
7	An example showing the feasibility of verifying inconsistencies based on data extracted from different levels . . . . .	25
8	The motivating example . . . . .	27
9	CiT System Overview . . . . .	30
10	An example of causal dependencies between events in Tacker/OpenStack services . . . . .	33
11	Example Tacker log entries and the output of data pre-processing (event sequences and corresponding parameters) . . . . .	35
12	An example of Seq2Seq translation using the LSTM Encoder-Decoder model for an event sequence from the VNF level to the NFVI level . . . . .	37

13	An example of training the Siamese Manhattan Network with NFV event sequences . . . . .	40
14	An example of event-level and workflow-level inconsistency detection . . .	41
15	A real-world NFV deployment on our testbed implemented using OpenStack Tacker and ODL. The circled numbers indicate deployment stages: 1) Onboarding the NS Descriptors, 2) Deploying the VNFs, 3) Configuring the VNFs and 4) Instantiating the NS . . . . .	45
16	VNF catalog in Tacker Horizon showing a list of onboarded VNFDs . . . .	48
17	VNFM module in Tacker Horizon showing a list of deployed VNFs . . . .	48
18	VNFFG catalog in Tacker Horizon showing a list of onboarded VNFFGDs .	49
19	VNFFG manager module in Tacker Horizon showing a list of deployed VNFFGs . . . . .	49
20	OpenStack VIM . . . . .	50
21	OpenStack system overview . . . . .	50
22	OpenStack system defaults . . . . .	51
23	A list of all the OpenStack services in the NFV testbed . . . . .	51
24	OpenFlow Rules corresponding to the deployed VNFFGs . . . . .	52
25	(a) The growth of vocabulary size, and (b) the proportion of unseen event types . . . . .	57
26	Visualization of NFV events with t-SNE . . . . .	58
27	The ROC evaluation results of $CiT_s$ based on D1 . . . . .	60
28	The ROC evaluation results of $CiT_s$ based on D2 . . . . .	61
29	The ROC evaluation results of $CiT_s$ based on D3 . . . . .	61
30	The ROC evaluation results of $CiT_s$ based on D4 . . . . .	62
31	The ROC evaluation results of $CiT_s$ based on D5 . . . . .	62
32	The ROC evaluation results of $CiT_s$ based on D6 . . . . .	63

33	The ROC evaluation results of $CiT_s$ based on D7 . . . . .	63
34	The ROC evaluation results of $CiT_s$ based on D8 . . . . .	63
35	The ROC evaluation results of $CiT_s$ based on D9 . . . . .	64
36	The ROC evaluation results of $CiT_s$ based on D10 . . . . .	64
37	The ROC evaluation results of $CiT$ based on the datasets D1 . . . . .	66
38	The ROC evaluation results of $CiT$ based on the datasets D2 . . . . .	66
39	The ROC evaluation results of $CiT$ based on the datasets D7 . . . . .	66
40	The ROC evaluation results of $CiT$ based on the datasets D9 . . . . .	67
41	The ROC evaluation results of $CiT$ based on D10 tested on D9 . . . . .	68
42	The impact of number of epochs on $CiT$ - AUC vs. # of epochs . . . . .	71
43	The impact of number of epochs on $CiT$ - Loss vs. # of epochs . . . . .	71
44	The impact of number of epochs on $CiT$ - Accuracy vs # of Epoch . . . . .	72
45	The impact of number of epochs on $CiT$ - Precision vs # of Epoch . . . . .	72
46	The impact of number of epochs on $CiT$ - Recall vs # of Epoch . . . . .	72
47	Evaluation of data separation ratio - AUC . . . . .	73
48	Evaluation of data separation ratio - Loss . . . . .	73
49	Evaluation of data separation ratio - Training Time . . . . .	74
50	$CiT$ efficiency study: Training Time. The results are obtained based on LSTM with $E_m = 256$ and $U = 250$ . . . . .	77
51	$CiT$ efficiency study: Time vs. Training Pairs. The results are obtained based on LSTM with $E_m = 256$ and $U = 250$ . . . . .	77
52	$CiT$ efficiency study: Hidden Unit Type. The results are obtained based on LSTM with $E_m = 256$ and $U = 250$ . . . . .	78
53	$CiT$ efficiency study: Event Embedding Dimension; The top sub-figure shows the results from D9 dataset, and the bottom sub-figure shows the results from D10 dataset. . . . .	78

54	<i>CiT</i> efficiency study: Seq. Embedding Dimension; The top sub-figure shows the results from D9 dataset, and the bottom sub-figure shows the results from D10 dataset. . . . .	78
55	The evaluation of testing time . . . . .	80
56	The topology view of VNFs implemented in our NFV tested from Horizon [66] . . . . .	90

# List of Tables

1	Main acronyms used in this thesis . . . . .	7
2	Dataset statistics (the gray shaded datasets are processed real data) . . . . .	53
3	Statistics of the original real data (from May 2017 to March 2020) . . . . .	53
4	Robustness evaluation of <i>CiT</i> : Sequence label translation for the real-world data. (%) indicates the percentage of correctly translated labels. . . . .	69
5	Robustness evaluation of <i>CiT</i> . Case study on real-world bugs and denied operations . . . . .	70
6	The evaluation of data separation based on five metrics . . . . .	74
7	AUC (%) vs. event embedding dimensions ( $U = 256$ ) . . . . .	75
8	AUC (%) vs. event sequences embedding dimensions ( $E_m = 250$ ) . . . . .	75
9	AUC (%) vs. network hidden unit types ( $U = 256/E_m = 250$ ) . . . . .	76

# Chapter 1

## Introduction

Network Functions Virtualization (NFV) has emerged as one of the main technology pillars of 5G networks [35, 112]; for instance, 60% of network service providers will be adopting NFV by 2021 [80] and the NFV market size is projected to grow from \$12.9B in 2019 to \$36.3B by 2024 [34]. The main benefit of NFV comes from its power in decoupling the network functions, such as firewall or intrusion detection, from dedicated and proprietary hardware appliances. By providing network functions through software-based Virtual Network Functions (VNFs) running on top of standard hardware infrastructures, NFV makes it possible for providers to deploy dynamic, agile, scalable, and cost-efficient network services.

### 1.1 Inconsistencies in an NFV Stack

Despite such advantages, the increased complexity of an NFV stack means the attack surface of NFV environments will be significantly larger than that of traditional networks, leading to novel security vulnerabilities and threats [30]. Existing works [26, 41, 83, 110] have addressed various security threats in NFV (e.g., vulnerabilities in VNFs, vulnerabilities due to orchestration and management complexities, and vulnerabilities resulting from

the lack of interoperability) and proposed corresponding solutions (e.g., hypervisor introspection, secure zoning, and image signing).

However, a security issue that is intrinsic to NFV has received little attention, i.e., as NFV environments typically involve several levels of abstraction, the inconsistency between those levels may arise due to the lack of proper synchronization between management and orchestration components, which can be exploited by malicious adversaries for security attacks. Although the inconsistency threats have been investigated in other contexts such as cloud and SDN [46], [109], it has only received limited attention in NFV [23, 82, 83, 94] and there lacks an in-depth study about how such inconsistencies may be instantiated and exploited based on concrete deployment of NFV, and how such inconsistencies may be modeled and identified based on existing data in NFV. Additionally, such attacks may cause severe security concerns, such as unauthorized modifications of Service Function Chains (SFCs), network eavesdropping, and DDoS, as evidenced by real-world vulnerabilities (e.g., [56–58, 71, 72, 74, 75]) and reported in recent studies (e.g., [23, 24, 41, 83, 90, 94, 110]). Therefore, to ensure the secure deployment of NFV, it is essential to detect such inconsistency attacks (in fact, our solution is able to detect many of the above-mentioned real-world vulnerabilities, as shown later in this work).

To that end, most of the existing works (e.g., [25, 27, 28, 47, 48, 50, 99, 102, 113, 114]) verify the state (e.g., configurations) of the NFV system to detect such inconsistency attacks, which suffers from the following limitations. First, those solutions are mostly retroactive in nature as they can only conduct the inconsistency verification after the fact and with a delay (e.g., time to verify). Therefore, those solutions cannot prevent irreversible damages (e.g., information leakage and DoS). Second, most of those approaches (e.g., [25, 28, 50, 102, 114]) heavily rely on the access to lower-level configurational data (e.g., network flow rules, flow classifiers, etc.). However, the practicality of those works could be limited when multiple providers are involved, as accessing the required lower-level data



may become infeasible [98]. Finally, to the best of our knowledge, none of the existing works considers the implication of all abstraction levels in the NFV stack, whereas most only focus on part of the NFV stack (e.g., the physical infrastructure [47], the virtual infrastructure [48], or Service Function Chaining (SFC) [27, 99, 102, 113, 114]).

## 1.2 Thesis Statement

In this work, we observe the gap between what is needed for understanding the inconsistencies (i.e., detailed information about the NFV deployment) and what is currently available in the ETSI NFV reference architecture [17]. The observation leads us to devise a novel NFV deployment model based on studying existing NFV deployment in open source platforms. Our deployment model complements the ETSI NFV architecture with details about all the critical components of an NFV environment, their relationships, and their levels of abstraction. Our deployment model enables us to present concrete attack scenarios in which the inconsistency vulnerabilities are exploited to attack NFV in a stealthy manner. We validate our model and attacks through implementation based on a real NFV testbed, and presenting a feasibility study on the verification solution by gathering information required for identifying the consistency.

As a solution for detecting such inconsistencies in an NFV stack, we propose a novel approach, namely, *Caught in Translation (CiT)*, to translate event sequences between different levels of an NFV stack and detect cross-level inconsistency attacks. More specifically, we first study the cross-level mapping of NFV events and generate embeddings for both events and event sequences. Second, we devise an NMT-based technique (tested with Long Short-Term Memory (LSTM) [32], Gated Recurrent Unit (GRU) [14], and simple Recurrent Neural Network (RNN) [3]) to translate event sequences between different levels of NFV. Finally, we apply both Siamese neural network [55] and traditional similarity

metric [91] to quantify the similarity between a translated event sequence and the actual event sequence in order to detect any inconsistencies. We implement *CiT* and integrate it into OpenStack/Tacker [98], a popular choice for NFV deployment on cloud management platform. Through extensive experiments with both real and synthetic data, we demonstrate the accuracy and efficiency of our approach. Finally, we discuss several use-cases to demonstrate the practicality of *CiT*.

### 1.3 Contributions

In summary, the main contributions of this work are as follows:

1. To the best of our knowledge, our NFV deployment model is the first effort to capture how NFV is deployed in the real world based on open source platforms, and we believe such a model may see many other applications.
2. The attack scenarios demonstrate both the feasibility and the severity of inconsistency-based security threats, which could draw more attention to this issue and provide insights to its mitigation. Our study about the information required for identifying inconsistencies serves as a foundation for developing security verification solutions to detect such threats.
3. To the best of our knowledge, *CiT* is the first event-based approach to detect inconsistency attacks in an NFV stack. In contrast to most after-the-fact approaches, *CiT* can catch malicious events at runtime before such events cause any potentially irrecoverable damages, such as the leakage of sensitive information or denial of service.
4. *CiT* demonstrates the potential of a translation-based detection approach. First, *CiT* is shown to outperform traditional anomaly-based detection in terms of accuracy. Second, the comparison of the three variations of *CiT* shows that translation can

significantly improve the detection accuracy. Finally, the translation capability may have other applications in NFV, such as providing translated events at a level where the access to actual events is prohibited (e.g., by a different provider).

5. The practicality of *CiT* is demonstrated through its integration into OpenStack/Tacker. Its accuracy and efficiency are evaluated through extensive experiments using both real and synthetic NFV datasets, and its robustness is examined through training and testing with data collected from different systems, and capturing anomalous events triggered by real world bugs and errors. Additionally, we discuss the extension of *CiT* to other NFV platforms (e.g., OSM [78], and OPNFV [77]).

## 1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 reviews the ETSI NFV architecture and introduces our NFV deployment model. Chapter 3 presents the attack scenarios and studies the feasibility of consistency verification. Chapter 4 introduces our translation-based inconsistency detection system. Chapter 5 details the implementation and provides the experimental results. Chapter 6 provides more discussions. Chapter 7 reviews the related work and Chapter 8 details contributions to other projects. Chapter 9 elaborates the future work and concludes the thesis.

# Chapter 2

## NFV Deployment Model

In this chapter, we first review the ETSI NFV architecture and explain what additional information is needed to understand the inconsistencies through a motivating example. We then introduce our multi-level NFV deployment model.

### 2.1 Preliminaries

#### 2.1.1 The ETSI NFV Reference Architecture

ETSI [19] introduced the NFV reference architecture as represented in Figure 1, to enable dynamic deployment and management of VNF instances and the relations between them [19]. Figure 1 shows the NFV reference architecture from ETSI (the callouts are not a part of ETSI NFV architecture and will be explained in Chapter 2.2). The architecture includes three main blocks, namely, Virtual Network Functions (VNFs), NFV Infrastructure (NFVI), and NFV Management and Orchestration (MANO). First, VNFs provide a high-level representation of network functions. Second, NFVI represents the cloud infrastructure that provides basic compute, network and storage capabilities. Third, MANO supports dynamically managing and orchestrating the lifecycle of physical and virtual resources,

which is further divided into three managerial components, Virtual Infrastructure Manager (VIM), Virtual Network Function Manager (VNFM), and Network Function Virtualization Orchestrator (NFVO), to complete the entire deployment process. The Operating Support System/Business Support System (OSS/BSS) is considered as an independent module supported by MANO.

Table 1 lists the main abbreviations we use in this thesis.

Acronym	Full Form	Acronym	Full Form
CP	Connection Point	PPG	Port Pair Group
EMS	Element Management System	RNN	Recurrent Neural Network
FC	Flow Classifier	SDN	Software Defined Networking
GRU	Gated Recurrent Unit	SDN-C	SDN Controller
LSTM	Long Short-term Memory	SFC	Service Function Chain
MANO	Management and Orchestration	SVM	Support Vector Machine
NFP	Network Forwarding Path	TFIDF	Term Frequency-inverse Document Frequency
NFV	Network Function Virtualization	VDU	Virtual Deployment Unit
NFVI	Network Function Virtualization Infrastructure	VIM	Virtual Infrastructure Manager
NFVO	Network Function Virtualization Orchestrator	VM	Virtual Machine
NMT	Neural Machine Translation	VNF	Virtual Network Function
NS	Network Service	VNFD	VNF Descriptor
NSD	Network Service Descriptor	VNFFG	VNF Forwarding Graph
OOV	Out-of-Vocabulary	VNFFGD	VNFFG Descriptor
OvS	Open vSwitch	VNFM	VNF Manager

Table 1: Main acronyms used in this thesis

## Virtual Infrastructure Manager

VIM manages physical resources and the virtualized pool of compute, storage and network resources in NFVI. VIM interacts with NFVO and SDN controller to facilitate network service orchestration and traffic steering respectively. VIM's northbound interface facilitates orchestration and management module of NFV. The southbound interface interacts with the hypervisor and network controller to implement virtualization and traffic steering respectively.

## Virtual Network Function Manager

VNFM is responsible for performing the life-cycle operations (for e.g., instantiate, scale, notify, alert, terminate) of the VNFs.

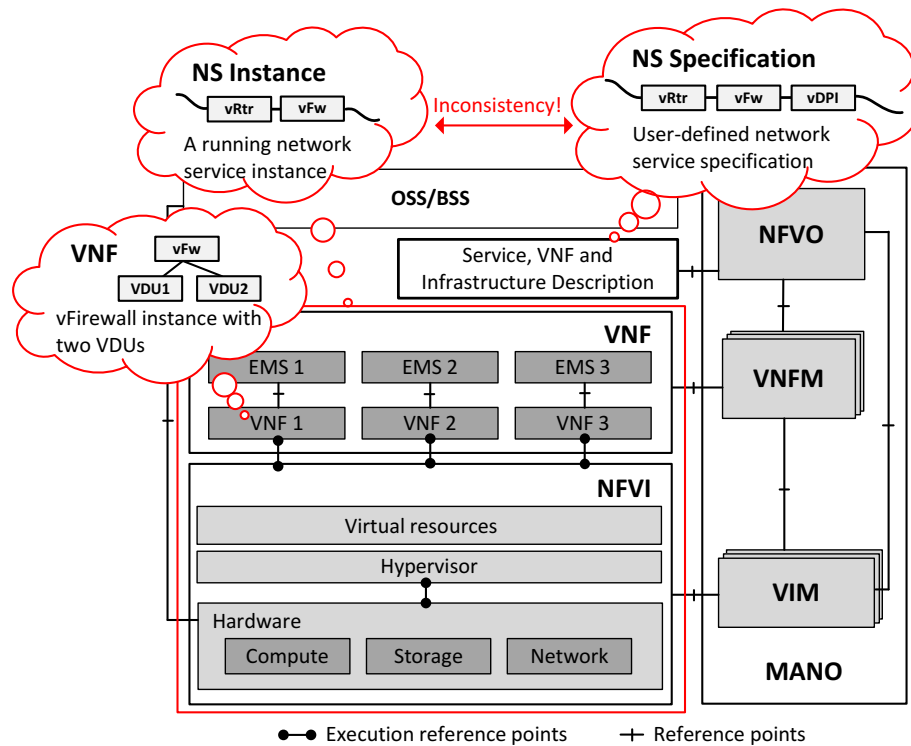


Figure 1: The ETSI NFV reference architecture [19]

## Network Function Virtualization Orchestrator

NFVO handles the complete deployment process of an NS. The two main functionalities of NFVO are: (i) *Resource Orchestration*, where it orchestrates the resources under the control of multiple VIMs and (ii) *Service Orchestration*, where it coordinates with multiple VNFMs to create and manage an end-to-end NS. Hence, NFVO is often referred as “the brain of NFV”.

Hence, MANO can be considered as the most critical block inside this architecture in terms of its role in inducing inconsistencies. For example, if the three managerial components are in accordance with each other for every change inside the NFV environment, inconsistency can certainly be avoided up to a degree. However, that is not the case according to the current implementation methodologies. Therefore, it is important to understand the role of each managerial component before exploring the inconsistency problem.

### **2.1.2 NFV Service Models**

The following NFV use cases are proposed by ETSI which corresponds to the existing cloud service models (IaaS, PaaS and SaaS) [18].

#### **NFV Infrastructure as a Service (NFVIaaS)**

This model enables cloud service providers to lease the infrastructure from another service provider (e.g., a Network Service Provider) to deploy their virtual NSs. Network service provider shall be able to integrate their VNF instances running on the leased infrastructure into an end-to-end network service instance on their own NFV infrastructure.

#### **Virtual Network Platform as a Service (VNPaaS)**

In this case, the service provider builds a generic virtual platform which is used by an enterprise to deploy their own network services. This model allows the enterprises to have complete administrative privileges over the network service.

#### **Virtual Network Function as a Service (VNFaaS)**

VNFaaS is in accordance with Software as a Service (SaaS) model where a VNF is a network service provider's application and an enterprise (end-user) becomes the consumer of the network service. The end-user is only able to configure the VNFs without being able to control or manage the underlying platform.

### **2.1.3 The Cloud Architecture**

NIST defines cloud architecture into two distinct layers [59]. The *Management Layer* comprises of cloud management operations (e.g., create network), which assists tenants to manage their virtual environment. The *Infrastructure Layer* includes underlying implementations (e.g., VXLAN tagging, flow table management, traffic steering), which is the

key to implement multi-tenancy. SDN can be integrated into the cloud architecture to further define the east-west traffic flow, which provides the basis to realize service function chaining.

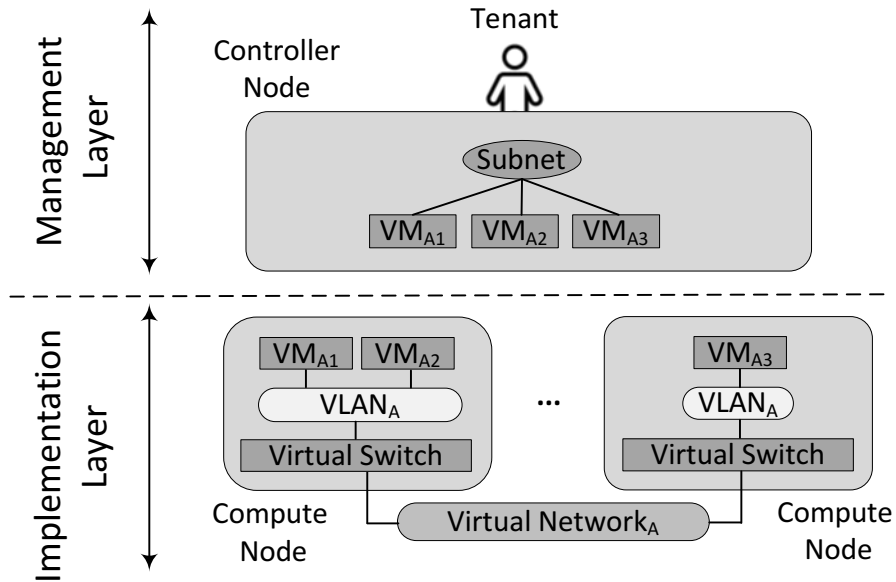


Figure 2: The NIST cloud model [59] depicting the two-layer view of a multi-tenant virtualized infrastructure in cloud

## 2.2 Motivating Example

To illustrate what might be missing in the ETSI NFV architecture when it comes to studying the inconsistencies, Figure 3 shows a simple example of inconsistency. First, the NS specification (top of the figure) shows that Bob has specified a virtual firewall (vFW) with two Virtual Deployment Units (VDUs), i.e., VDU2 with pfSense for routing and firewalling (the rule shows that any SSH requests should be rejected), and VDU3 with Snort for IDS. Second, the corresponding NS instance depicts the changing state of VDU2 before and after an attack is launched by another user, Alice. By exploiting a VM hopping vulnerability (e.g., CVE-2015-3456 (Venom), CVE-2015-7835, and CVE-2018-10853), Alice



gains control of VDU2 and modifies its pfSense rule to allow SSH requests to the Web server. Importantly, such a change made by Alice on VDU2 will not be reflected at the higher level (in the VNF Descriptor), which leads to a stealthy attack caused by the inconsistency between the two levels.

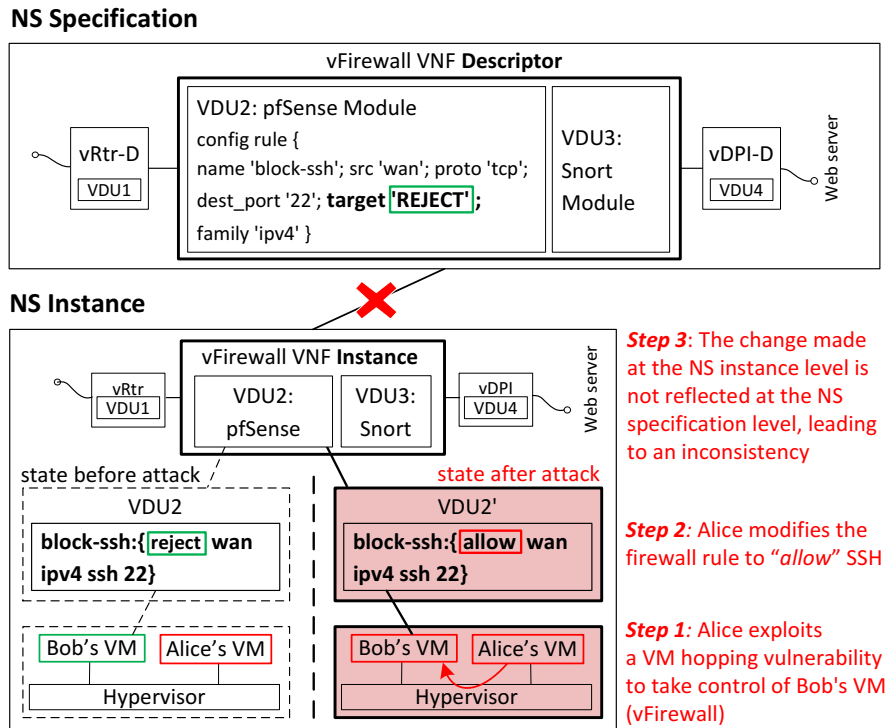


Figure 3: An inconsistency between the NS specification and instance

To model this attack using the ETSI architecture, we revisit Figure 1. As the callouts show, the attack involves the following NFV deployment details missing in the ETSI architecture.

- The mappings between VDUs and VNFs, NS instance and VNF, NS instance and NFVI are all absent from the ETSI architecture, e.g., the mapping between the virtual firewall descriptor and the two VDUs is essential to understand the inconsistency in our example.

- The mapping between the management and implementation layers of clouds [59] is also missing in the ETSI model, e.g., the mapping between Bob's VM and VDU2 allows us to link the attack *step 1* and *step 2*.
- Other details like the traffic steering related to the dependencies between the managerial components and the corresponding virtual resources are also missing.

To complement the ETSI architecture with such missing details, the next chapter devises an NFV deployment model.

## 2.3 The NFV Deployment Model

This chapter proposes a multilevel deployment model for NFV environments. We first provide an overview and then detail each level of the model.

### 2.3.1 Overview

Our NFV deployment model is based on the NFV deployment of several popular open source platforms including Open Networking Automation Platform (ONAP) [61], Tacker [64], OpenStack [64], and OpenDaylight [62]. We extract the operational dependencies between managerial components (i.e., NFVO, VNFM, VIM, and SDN-C) and functional elements (e.g., VNF, VM, SFC, and virtual switches). We separate the NFV stack into four levels as follows. The first level is based on the common deployment aspects from both ONAP and Tacker, i.e., NFVO and VNFM collaborate together to process descriptors and perform high-level management of VNFs and their connectivity. The second and third levels are based on NIST's cloud architecture [59], i.e., the management layer for cloud management operations and the implementation layer for underlying implementations.

Figure 4 presents our multilevel NFV deployment model (middle) with the mapping to the ETSI NFV reference architecture (left) and our motivating example (right). Specifically,



4. Finally, the *Physical Infrastructure* (L4) level depicts all physical resources to complete an end-to-end NFV stack.

### 2.3.2 L1: Service Orchestration Level

L1 includes the deployment components such as NFVO, VNFM, NSD, VNF, and EMS. Two key NFV managerial components at L1 are NFVO and VNFM, which manage NSs in the form of catalogs such as network service descriptor (NSD). NSD can include other descriptors, such as VNF descriptors (VNFD) and VNF forwarding graph descriptors (VNFFGD). Descriptors provide all necessary network service specifications and implementation information in structured templates for orchestration. For example, the traffic steering is defined in VNFFGD; the incoming network traffic would first pass flow classifiers (FC) before being forwarded to a specific network function path (NFP). An NFP connects VNFs with connection points (CPs) using virtual links.

To deploy an NS instance, an NFV user specifies the descriptors as inputs for NFVO. After NFVO validates the technical accuracy of such inputs, NFVO and VNFM inform VIM to allocate the underlying resources to implement the NS. NFVO manages the VNFFG (network topology), whereas VNFM with Element Management System (EMS) performs high-level management of the individual VNFs based on users' specifications, e.g., the logical mapping between VDUs and VNFs. However, the implementation details at lower levels are not reflected in VNFM. We place VNFs at the same level of VNFM because of this operational dependency.

**Example 1.** Tenant Bob wants to deploy an NS with three VNFs to steer traffic to two destinations. The right side of L1 in Figure 4 shows this NS is deployed using four descriptors. The NSD for this instance is the composition of VNFFG1D,  $vRt rD$ ,  $vFwD$ , and  $vDPID$ . The lower part of this sub-figure bridges the descriptors with the implementation. For example,  $vFw$  at this level includes the interpretation of its corresponding descriptor

$vFwD$  and the instance identifiers of  $VDU2$  and  $VDU3$  from the lower level. This logical mapping helps to understand the deviation between  $vFwD$  and  $VDU2$  should any inconsistencies occur. Logically,  $NFP1$  contains the sequential order of connection points (e.g.,  $CP:vRtr$ ) to chain all three VNFs.  $FC1$  classifies all HTTP traffic to  $NFP1$ , while  $FC2$  sends the management traffic through  $NFP2$ .

### 2.3.3 L2: Resource Management Level

L2 contains virtual resources such as VDUs, subnets, SFC, network ports, etc., depicting how the NFV virtual resources are created and managed inside the cloud environment. VIM is directly responsible for provisioning, interconnecting and decommissioning these virtual resources contained in an NFVI-PoP domain (an NFVI instance). VDUs at L2 follow a many-to-one relationship with the VNFs from L1 and a one-to-one logical mapping with the VMs from the lower level. VNFFGs from L1 are instantiated as service function chains (SFCs; also referred to as port chains). SFC is a sequence of port pair groups, which consist of one or more port pairs.

Once the resource allocation request is received from NFVO, VIM allocates the compute, storage and network resources corresponding to the given descriptors. Then, the virtual resources, such as VDUs, subnets, network ports, routers, service chains, are instantiated to build an NS. Although VIM can be considered as a part of both L2 and L3, the management operations are executed by cloud tenants from L2 through VIM to directly manage the virtual resources. Hence, VIM is considered as a part of L2 in our model.

**Example 2.** Once VIM receives the resource creation request from NFVO, it creates the deployment units (e.g.,  $VDU1$ ) as shown on L2. These VDUs are then configured based on NSD. For example, the mapping of  $vFw$  to  $VDU2$  and  $VDU3$  is configured to allow HTTP traffic and block SSH traffic. After the creation and configuration of VNFs, the network

paths are created as defined in VNFFGD. These function paths are implemented as port-chains `nfp1-chain` and `nfp2-chain` with the respective flow-classifier instances `fc1` and `fc2`. By definition, `nfp1-chain` is an ordered list of port-pair-groups (CPs at L1) `PPG1`, `PPG2` and `PPG3` corresponding to the virtual instances `VDU1`, `VDU2` and `VDU3`, respectively.

### 2.3.4 L3: Virtual Infrastructure Level

L3 includes SDN-C and the virtual networking elements, such as virtual switches, VLANs, VxLANs, virtual routers, flow tables, virtual bridges, and the corresponding interfaces. SDN-C is the management element inside this level (we follow the most widely adopted use case in the SDN usage specification [20] to place SDN controller at this level for providing network connectivity). Flow-tables form the fundamental elements for network traffic steering. These flow-tables are populated with flow-rules, which depict the forwarding behavior of the SFC. Once the service chains are created, the corresponding traffic flows for each chain are deployed in the virtual switches for traffic steering through VMs. The deployment of traffic flow rules is carried out by the SDN-C; this is another reason for us to place SDN-C at this level.

Listing 2.1: Flow-rule that forwards HTTP traffic to vRtr VNF

```
cookie=0x794188fe368fe901 , duration=1108281.174s , table=0 , n_packets=0 , n_bytes=0 , priority=30,tcp , in_port="qvo6094f47a-3f" , nw_src=192.168.100.11 , nw_dst=192.168.100.100/32 , actions=group:1
```

**Example 3.** Listing 2.1 shows the flow-rule that classifies and forwards the HTTP traffic through `vRtr` to the web server. `table0` acts as an FC for the SFC. Each flow-rule has a *match* criteria followed by the *action* that is to be performed for the matched traffic. In case of `nfp1-chain`, simple *match* criteria is to select all HTTP packets originating from the

*in\_port*: qvo6094f47a-3f with the *nw\_src* as 192.168.100.11 and the destination as 192.168.100.100/32. Then the *action* is to forward the matched packets to vRtr VNF, which is denoted as *group:1* referencing the VNF's port-pair-group. The traffic is then forwarded to *group:2* (vFw) and *group:3* (vDPI) before it reaches the actual destination.

### **2.3.5 L4: Physical Infrastructure Level**

This level includes all the physical entities (e.g., COTS servers as controller and compute nodes, and physical network functions (PNFs)) that are involved in the NFV stack. The actors involved at this level are typically the physical infrastructure operators or facility managers who also represent the managerial components. The model does not include many details at this level for the sake of clarity.

# Chapter 3

## Inconsistencies in NFV Stack

In this chapter, we first discuss inconsistencies in NFV and explore potential attacks for exploiting the inconsistencies at different levels. We then provide two concrete attack scenarios to validate our multi-level NFV deployment model.

### 3.1 Overview

The inconsistency between the NS specification and instance as discussed in our motivating example (Chapter 2.2) is only a special case. Despite the fact that NFVO is considered as the “brain” of an NFV environment, the other managerial components can operate at each level autonomously, which is referred to as the “split-brain” issue in the literature [21]. For example, VIM and SDN-C can manipulate virtual resources and virtual network freely without going through NFVO. Such autonomous management is intentional in order to effectively manage multiple domains in a single NFV environment (e.g., there can be multiple VIMs managing many NFVI Points of Presence (NFVI-PoP)). However, the lack of synchronization is not intended [82], and it can lead to inconsistencies whenever the states of functional elements managed by two different managerial components differ from each



other. Additionally, in order to facilitate VNFs to mimic the promiscuous mode of traditional networking devices and to route traffic, port security (i.e., packet filtering on a port) is disabled for the VNF instances during NFV deployment by default. The downside of this feature is that, there is no network isolation between tenants who are on the same network. These fundamental design properties of NFV greatly facilitate the attacks that lead to significant discrepancies in the overall system. Aftermost, the NFV client who is dependent on the underlying levels is unaware of the contemporary functionality of his own network service. These inconsistencies can potentially lead to unexpected behaviors or network errors, security threats [83, 94] including some stealthy attacks as presented later in this chapter.

Therefore, the inconsistency may potentially arise between any managerial components and their functional elements inside an NFV environment. To that end, our multilevel NFV deployment model provides a foundation for analyzing potential inconsistencies, as it sufficiently captures the relationships between different levels of components in an NFV environment. For example, the inconsistencies between the management level and the implementation level of cloud [46] and SDN [109] could be mapped to L2 and L3, while the inconsistencies between user specification and the actual deployment could be captured by comparing L1 to the lower levels.

Next, we investigate potential attacks exploiting the cross-level inconsistencies based on our deployment model shown in Figure 4 and a concrete implementation based on OpenStack Tacker [64] and OpenDaylight (ODL) [62] (which will be detailed in Chapter 5). We will focus on possible attacks originated at L2 and L3, respectively, which could cause inconsistencies with the user specifications given at L1.

## 3.2 Threat Model

We assume that the adversary can be a malicious cloud tenant, an admin operator, or an external attacker who controls a virtual machine (e.g., via malware infection) with system privilege. The adversary is assumed to share part of the infrastructure (e.g., compute hosts and physical network) with the victim. We also assume that the cloud infrastructure can have vulnerabilities that the adversaries may identify and exploit. We do *not* assume the adversaries can compromise the managerial elements, and we do *not* assume the adversaries can compromise SDN controllers or switches.

## 3.3 Attacks targeting the consistency property

This chapter explains the possible attack cases which can lead the NFV stack to an inconsistent state.

### 3.3.1 Attacks at L2

By targeting the managerial component in this level (VIM) and the other functional elements such as VDU, port-chain, security groups etc., an attacker gains a variety of opportunities to render the NFV system to an inconsistent state.

The potential attacks originated at this level could lead to the modification of VDUs, port-chain, security groups, etc., which could all lead to an inconsistent state of the NFV system. We discuss several possibilities to achieve such attacks in the following.

- Through VIM, a malicious cloud admin, a cloud operator colluding with an external attacker, or a malicious cloud user exploiting a privilege escalation vulnerability will be able to modify the functional elements of another user's NS. Over time, OpenStack has seen several privilege escalation and sensitive data exposure vulnerabilities, such

as OSSA-2016-005 and OSSA-2017-004 [68]. By exploiting such vulnerabilities, an attacker can perform many unauthorized operations, such as updating a service chain by including a malicious VNF, for the system to reach an inconsistent state.

- As illustrated in our motivating example, an attacker can also take over the control of a VM through exploiting a hypervisor vulnerability and then modifying either its configurations or the traffic flow to lead to an inconsistent state in which the attacker's actions at L2 are not reflected at L1 causing a stealthy attack.

### 3.3.2 Attacks at L3

Existing security threats in SDN, such as malware infection, topology poisoning [33], control plane saturation [95], and state manipulation attacks [107], can be employed to manipulate the traffic flow in L3. We discuss some possible attacks at this level as follows.

- An attacker can send crafted packets to the SDN controller to externally trigger undesirable events that lead to inconsistencies. For example, by enabling/disabling the network interfaces on a host, the attacker can trigger host-related events such as *HOST\_JOIN*, *HOST\_LEAVE*, etc.
- An attacker can compromise a virtual switch and program it to modify traffic flows to cause inconsistencies. For example, a critical vulnerability (*CVE-2018-1078* [63]) in ODL can be exploited to cause uncontrolled communication between VNFs by programming the switch to reconnect to the network upon new flow update events.

## 3.4 Attack Scenarios

This chapter introduces two real attack scenarios which were implemented in our NFV testbed (introduced in Chapter 5) highlighting the feasibility of an NFV stack to be led to an inconsistent state.

### 3.4.1 Attack Scenario 1

Using our testbed, we have implemented a concrete attack that targets the integrity of a service function chain (SFC).

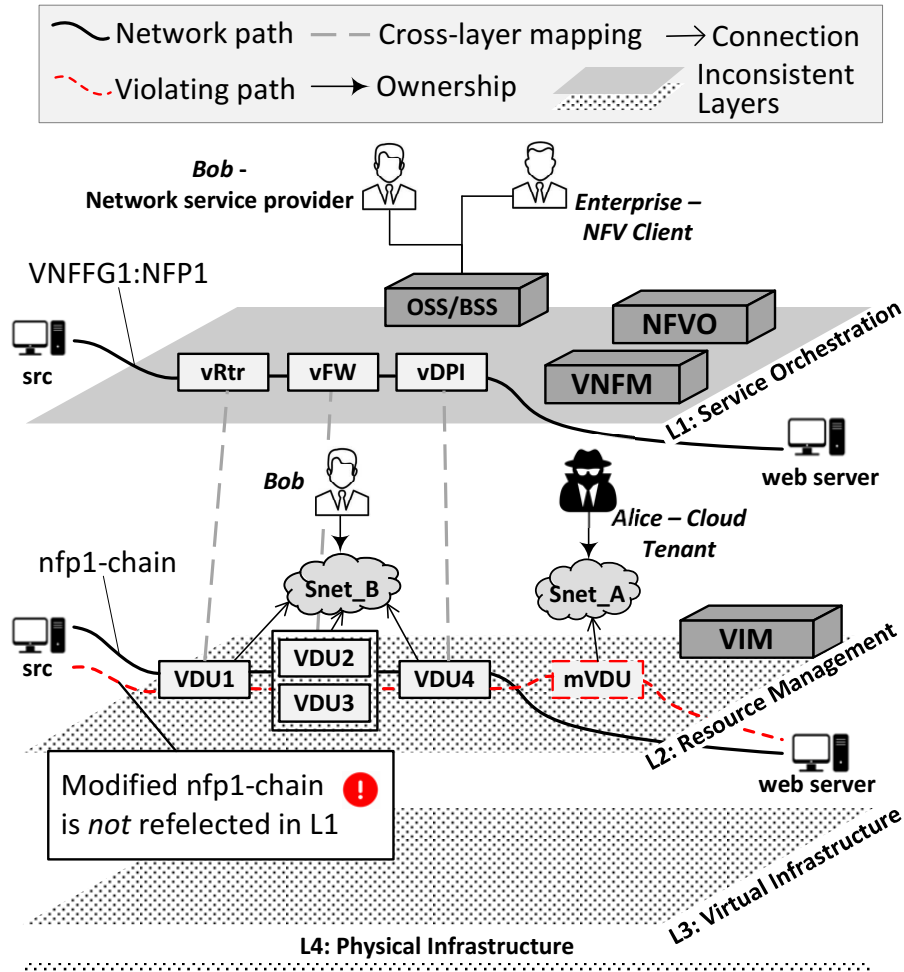


Figure 5: An implemented attack at L2 causing inconsistency between the path specification VNFFG1 :NFP1 and its instance nfp1-chain

In Figure 5, Bob is a network service provider serving enterprise NFW clients who happens to share the physical infrastructure with a malicious tenant Alice. The red dashed line shows a compromised service chain instance, nfp1-chain, which is modified by Alice to include a malicious VNF. However, as our test has shown, such a modification at L2 will not be reflected at L1, leading to an inconsistent state of the NFW stack and a

stealthy attack allowing Alice to inspect or modify traffic passing the chain.

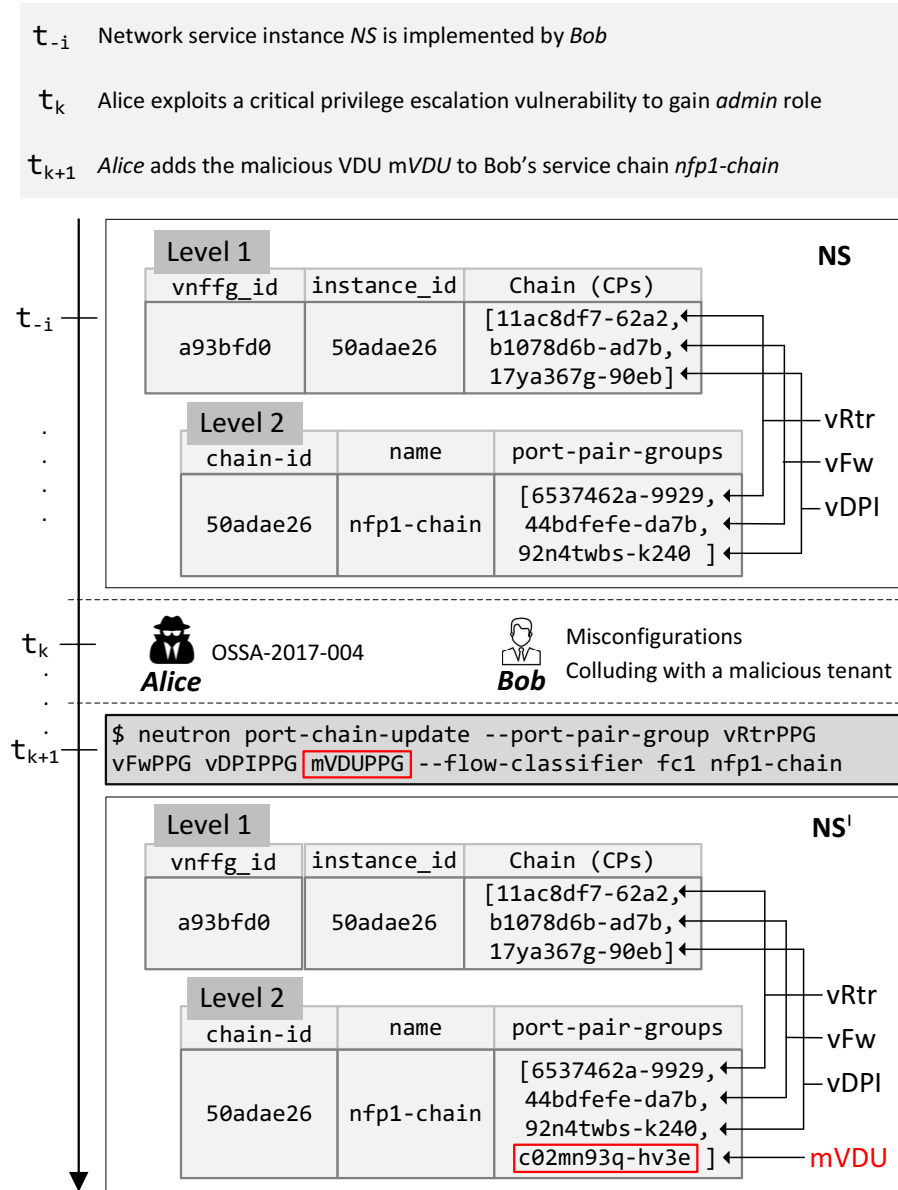


Figure 6: An illustration of the attack timeline when Alice modifies Bob's port-chain *nfp1-chain* by adding a malicious *mVDU* at L2

More specifically, Figure 6 shows the attack timeline. The port-chain instance (*nfp1-chain*) of NFP1 consists of three port-pair-groups corresponding to the three VNFs at  $t_{-i}$  time. Alice at  $t_k$  could perform the aforementioned attack to execute the *neutron port-chain-update*

command which would update `nfp1-chain` by adding the port-pair-group of a malicious VNF `mVDU`. Upon the execution of the `neutron port-chain-update` command, the flow-rules will be updated in the virtual switches for `vDPI` to forward the traffic further to `mVDU` as opposed to the path definition in the `VNFFGD` element of `L1`. To this end, the `mVDU` VNF is added as a hop in Bob's port-chain allowing Alice to have unauthorized access to any traffic flowing through `NFP1` without being noticed.

### 3.4.2 Attack Scenario 2

The second concrete attack targets the flow-tables at `L3` for causing inconsistencies between `L3` and the upper levels. When the network topology gets updated, the SDN controller will install new flow-rules in the virtual switches to reflect the changes. To manipulate the flow-tables at `L3`, an attacker can trigger a virtual switch reconciliation (a functionality to ensure that switches properly reflect intended controller configurations after restarts) by sending crafted network packets during a network topology update. This would cause the old flow-rules to be installed instead of the new flows. Therefore, the traffic will be steered as specified by the old flow definition contravening the topology update.

Specifically, assuming the end user (NFV Client) updates `NS`'s topology by changing its `VNFFG1` definition at `L1` to add `vDPI` to `NFP2`, which will redirect the management server's traffic through `vDPI` for further analysis. Upon the update, the corresponding port-chain (`nfp2-chain`) and flow-rules in the virtual switches must be updated at `L2` and `L3`, respectively. Meanwhile, an attacker triggers a `SWITCH_LEAVE` event by continuously resetting TCP sessions [107] between `compute1_vswitch` and the controller. When the switch reconnects, the old flow-rules are re-installed as a consequence of the node-reconciliation vulnerability, leading the traffic of `NFP2` to be forwarded to the management server directly without passing `vDPI`. However, `L1` and `L2` are not aware of this change in the actual traffic flow, leading the NFV system into an inconsistent state.

In addition to the attacks presented in this chapter, there might be many other ways for exploiting the cross-level inconsistencies in an NFV stack (e.g., attacking the flow-classifier component). To address this issue, we provide a feasibility study on identifying inconsistencies through verification.

### 3.5 Feasibility of Consistency Verification

Security verification using formal methods [44,94,109] or graph-based approaches [38] has seen applications in clouds and SDN, and it can also provide a viable solution for detecting the aforementioned inconsistencies in NFV. A key challenge is to understand what data needs to be collected from which component of an NFV stack in order to identify the inconsistencies. Based on our multilevel deployment model, we provide some preliminary results on the data collection to show the feasibility of verifying the consistency of NFV.

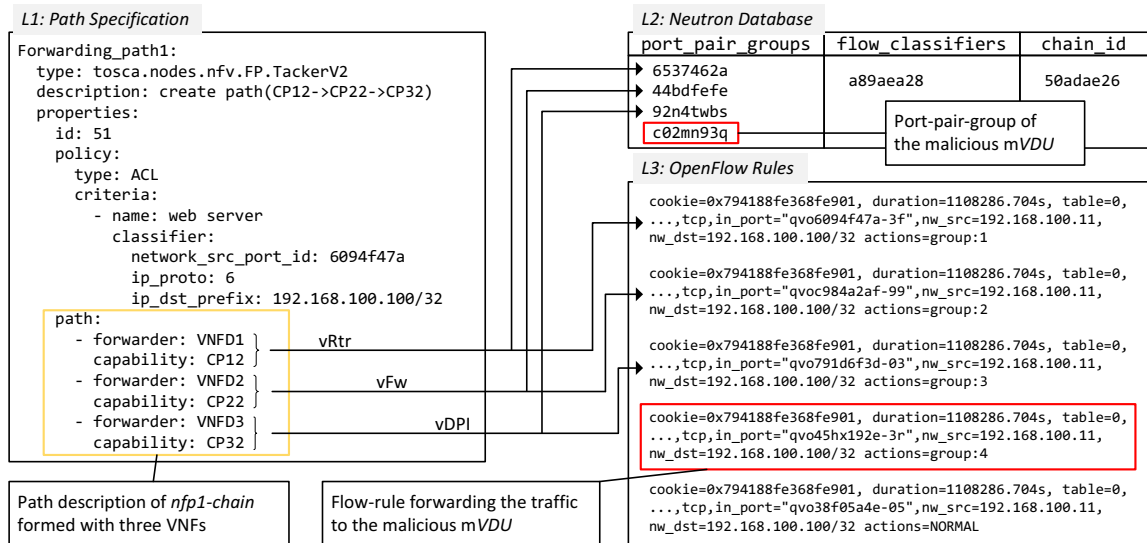


Figure 7: An example showing the feasibility of verifying inconsistencies based on data extracted from different levels

Specifically, Figure 7 shows an excerpt of the data sources within each level of our deployment model.

- *L1*: the descriptors at this level represent users’ requirements, which provide the baseline for verifying consistency. For example, NSD defines the path specification `nfpl-chain` with three VNFs (`vRtr`, `vFw` and `vDPI`) and the corresponding network path.
- *L2*: the information related to VNFs and SFCs can be extracted from VIM through several sources, such as Heat, Nova, Neutron databases, and the VDUs (e.g., VNF configurations and logs). The order of VNFs in a chain is preserved as the order of records in Neutron database, while the number of VNFs in a chain is corresponding to the number of records under the same `chain_id`. In our example, `chain_id 50adae26` contains four VNFs instead of three as defined in NSD. The inconsistency created by `mVDU` could be detected by comparing those data.
- *L3*: flow-tables from the virtual switches contain the information related to the forwarding behavior of NS. The highlighted flow-rule is the maliciously added flow, which forwards the traffic to `mVDU` (*group 4* in the rule), and also shows the inconsistency w.r.t. NSD.

More generally, the user-defined NS specification at L1 and the NS deployment related-data at L2 and L3 represent the current system state and will comprise the main inputs to verification mechanisms for detecting inconsistencies.



# Chapter 4

## Detecting Cross-level Inconsistency

### Attacks in NFV

Based on the NFV deployment model introduced in Chapter 2, we present *Caught-in-Translation (CiT)*, an LSTM-based sequence translation system for detecting inconsistency attacks in an NFV stack in this chapter.

#### 4.1 Motivating Example

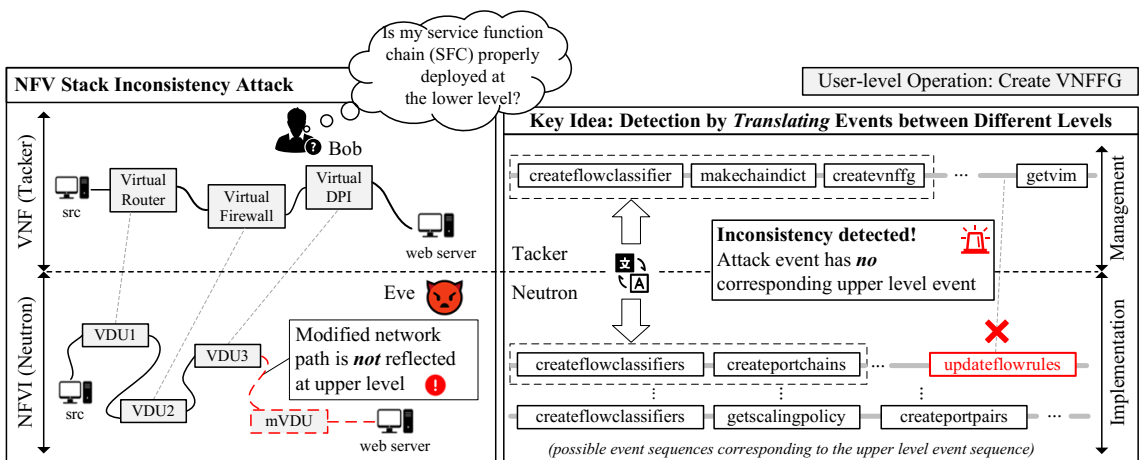


Figure 8: The motivating example

We present a motivating example to further highlight existing challenges in cross-level inconsistency detection for NFV and motivate towards our solution. The left side of Figure 8 shows the simplified NFV stack of a tenant Bob, which consists of an VNF level with three VNFs (*Virtual Router*, *Virtual Firewall*, and *Virtual DPI*) and a NFVI level with their corresponding Virtual Deployment Units (VDUs). For brevity, other virtual components, e.g., virtual machines, are omitted from the figure. Knowing that an adversary Eve could potentially inject a malicious VDU (*mVDU*) into Bob’s network directly at the NFVI level, without causing any noticeable changes to the VNF level <sup>1</sup>, Bob is concerned with the following question: “*Is my service function chain (SFC) properly deployed at the lower levels?*”

The right side of Figure 8 demonstrates the main ideas of our approach to detect inconsistency attacks. Specifically, a user-level operation *Create VNFFG* (i.e., creating a VNF forwarding graph) triggers two sequences of events at the VNF-level (in Tacker) and the NFVI level (in Neutron), respectively. Our key idea is to first *translate* the lower (Neutron) event sequence to the upper (Tacker) level, and then compare the translated event sequence to the actual Tacker event sequence, such that any inconsistency, such as the one shown in the figure (i.e., the Neutron event *updateflowrules* does not correspond to any Tacker event), can be detected. However, as shown at the bottom of the figure, a challenge here is that the same Tacker event sequence may correspond to many different Neutron event sequences, depending on the nature of events and the specification of the VNFs (e.g., a VNF may require multiple VDUs, and a VNF may have the autoscaling capabilities). Therefore, there does not exist a trivial mapping between the event sequences which would allow translation using simple rules. We will tackle this challenge using an NMT-based approach in the remainder of the thesis.

---

<sup>1</sup>For instance, by exploiting existing vulnerabilities CVE-2015-3456 [56], CVE-2015-7835 [57], or CVE-2018-10853 [58] in a specific way [96].

## 4.2 Key Ideas

In contrast to most existing works which mainly verify NFV configurations, our first idea is to take an event-based approach, i.e., examining and comparing events that occur at different levels of an NFV stack which correspond to the same user-level operation. The main advantage of relying on events instead of configurations is that we can potentially catch malicious events before they cause any irreversible damages. The events could be intercepted and checked (e.g., similar approaches exist for cloud platforms [4, 70]) to prevent malicious events from materializing their negative impact, e.g., injecting a malicious VNF into a Service Function Chain (SFC) to eavesdrop traffic flowing through the chain, or deleting legitimate VNFs from the chain to cause a denial of service. To that end, the main challenge is to detect the fact that certain events could cause inconsistencies between NFV configurations at different levels, *before* such inconsistencies are materialized. A straightforward solution could be to apply a standard anomaly-based detection mechanism, which will first train a model of normal events by considering event sequences at all levels of the NFV stack as the training data, and then utilize this model to identify any anomalous events causing inconsistencies. However, as evidenced by our experimental results, such an approach will be outperformed by our solution.

Our second idea is that we can draw an analogy between comparing event sequences at different NFV levels and translating sentences between different natural languages. As a result, we can leverage existing Neural Machine Translation (NMT) techniques designed for the latter to detect inconsistency attacks in NFV. Specifically, inside an NFV stack, a user-level operation, such as creating a VNF, will typically trigger an event sequence at each lower level. The fact that those lower-level event sequences all correspond to the same user operation means that their relationship is similar to the equivalence between sentences written in different natural languages. Thus, by leveraging this similarity to natural languages, we would be able to “translate” event sequences from one level to another, and

subsequently detect any inconsistencies by comparing the translated sequence to the actual one. Even though translation-based solutions (e.g., [15, 36, 85, 93, 105, 108, 115]) have already shown promising results in other domains (e.g., binary code similarity, network traffic anomaly, and Android malware detection), to the best of our knowledge, this is the first effort to apply such an idea in the context of NFV.

### 4.3 CiT Overview

This chapter first provides an overview of *CiT*, followed by the detailed methodologies of its major components.

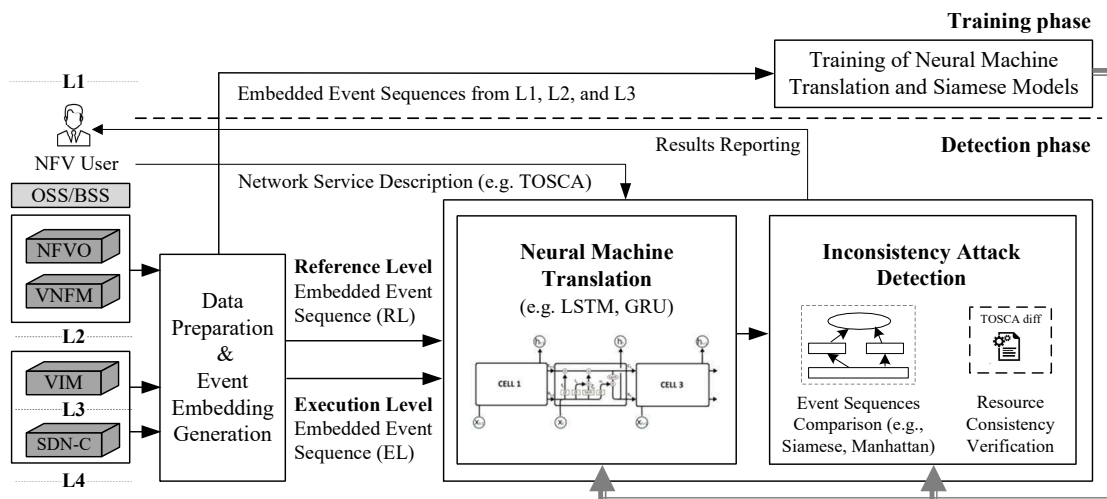


Figure 9: CiT System Overview

Figure 9 depicts how *CiT* prepares the data and trains the neural network models (training phase), and how it applies the trained models to first translate event sequences from one level of NFV to another, and then compare the event sequences to detect inconsistency attacks (detection phase).

### 4.3.1 Training Phase

This phase (top of Figure 9) mainly consists of the following three steps. First, *CiT* extracts events from all levels of the NFV stack, constructs sequences of events per level, and generates the embeddings of those event sequences (as explained in Chapter 4.4). Second, the NMT model [40] and the neural network similarity learning model [55] are trained based on the embeddings (as detailed in Chapter 4.5 and Chapter 4.6, respectively). Finally, the trained models are updated through retraining when any substantial change is made to the NFV system (e.g., major service updates which may potentially introduce new event types).

### 4.3.2 Detection Phase

This phase (bottom of Figure 9) mainly consists of the following steps. First, *CiT* generates embeddings for event sequences at both the *evaluated* level (from which inconsistency attacks are to be detected) and the *reference* level (to which the evaluated level will be compared). The next two steps will apply the trained models to translate and compare the event sequence from the evaluated level to the reference level in order to detect any inconsistency attacks. However, to support various use cases (see Chapter 6), *CiT* couples translation and detection in three different ways as follows.

- *CiT<sub>ts</sub>*: This variation of *CiT* first translates an embedded event sequence at the evaluated level into another embedded event sequence at the reference level, and then compares the similarities between those two embedded event sequences using neural network similarity learning (e.g., Siamese [55]). Finally, the similarity score is compared to a pre-defined threshold value to determine whether an attack has been detected.
- *CiT<sub>tm</sub>*: This variation first performs a similar translation step as *CiT<sub>ts</sub>*, and then compares the event sequences using a traditional similarity metric (e.g., Manhattan distance [1,91]) to detect any inconsistency.

- *CiT<sub>s</sub>*: This variation skips the translation step, and directly applies neural network similarity learning (e.g., Siamese [55]) on the collection of embedded event sequences from both levels to detect any inconsistency.

Finally, the resource-level inconsistencies are detected by generating and comparing TOSCA templates. We further elaborate on our methodology as follows.

## 4.4 Data Preparation and Event Embedding Generation

This chapter describes how *CiT* processes the raw data extracted from an NFV stack to construct event sequences and generate the embeddings (i.e., numerical representation) of those event sequences to support translation and detection.

### 4.4.1 Data Preparation

The main challenges in preparing NFV data for training are as follows.

- To build the training datasets correctly, we need to first understand the causal dependencies between events at different levels of the NFV stack as well as the mapping between event sequences across different levels.<sup>2</sup> This is challenging due to the complexity and the multi-level nature of the NFV stack.
- Raw events (either gathered from NFV service logs or intercepted at runtime using middleware plugins) typically contain many implementation specific details (e.g., platform-specific APIs) and parameters (e.g., request/resource IDs). Some of the non-essential details in such raw data, if fed directly into the training modules, could cause the Out-of-Vocabulary (OOV) challenge [11], a well-known problem in NMT.

Therefore, the raw data must be carefully processed in order to avoid the OOV issue

---

<sup>2</sup>Note that such understanding is only sufficient for preparing the training data and will not be exhaustive enough to be directly applied for actual translation.

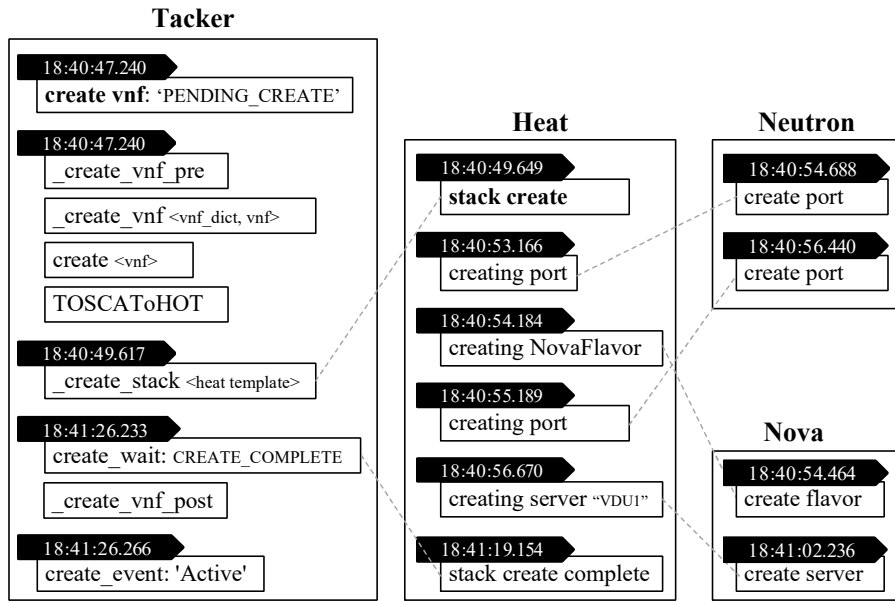


Figure 10: An example of causal dependencies between events in Tacker/OpenStack services

while preserving all the essential details (e.g., parameters) such that events belonging to the same sequence can still be identified even if they arrive out of order or interleaved.

- Unlike in NMT where models are trained once using large text corpora (e.g., WordNet [103] and Wiki) and there exist pre-trained models which can be reused (e.g., *GloVe* [84] or *fastText* [5]), we are the first to apply such techniques in the context of NFV, and therefore, we need to build our own domain specific corpus related to NFV events and train our models from scratch.

To overcome the first challenge, we studied NFV documentations and implementation details at different levels of the NFV stack. We have also deployed various network functions in our NFV testbed, and then collected and analyzed the logs to learn events and their order of occurrences within and across different levels using timestamps and resource parameters.

**Example 4.** Figure 10 shows an excerpt of the causal dependencies between different

events of various Tacker/OpenStack services, such as Tacker [98] (which is the VNFM in Figure 4), Heat [64] (which is the service orchestration engine), Neutron [64] (which is the network service), and Nova [64] (which is the compute service). The black labels indicate the timestamps of the events, and the dotted lines show the link between the events at different levels. The flow of events starts with a user-level operation (*create vnf* at the timestamp *18:40:47.240*), followed by a sequence of events in Tacker. After the *\_create\_stack* event in Tacker, the *stack create* event is triggered in Heat at the timestamp *18:40:49.649*, followed by a sequence of events in Heat. Later, as consequences of those events in Heat, the *create port*, *create flavor* and *create server* events are observed in the Neutron and Nova services, respectively. In the end, after the *stack create complete* event in Heat, Tacker finally generates the *\_create\_vfn\_post* event to complete the VNF creation.

To address the second challenge, our data preparation step first extracts the relevant events from different levels and aggregates those events into sequences corresponding to the same user-level operation based on their parameters, such as the resource ID and project ID, and timestamps. Afterwards, it strips out the implementation specific details and parameters from the sequences to avoid the out-of-vocabulary (OOV) problem, while keeping essential information such as the events' names before event embedding.

**Example 5.** Figure 11 shows an example of Tacker log records (top) and the corresponding processed sequences of events and extracted parameters (bottom). From the raw data, all events that correspond to the same user-level operation *create vnf* are identified, and classified into the following three stages to build a corresponding sequence of events: the initialization stage (i.e., *\_create\_vnf\_pre* and *\_make\_vnf\_dict*), the execution/creation stage (i.e., *\_create\_vnf*), and the confirmation stage (i.e., *\_create\_vnf\_post*).

In addition, even when the event sequences are consistent between all levels, finer-grained inconsistencies may still occur at the level of resource allocation (e.g., the implemented size of the memory is less than what is specified). Thus, our data preparation



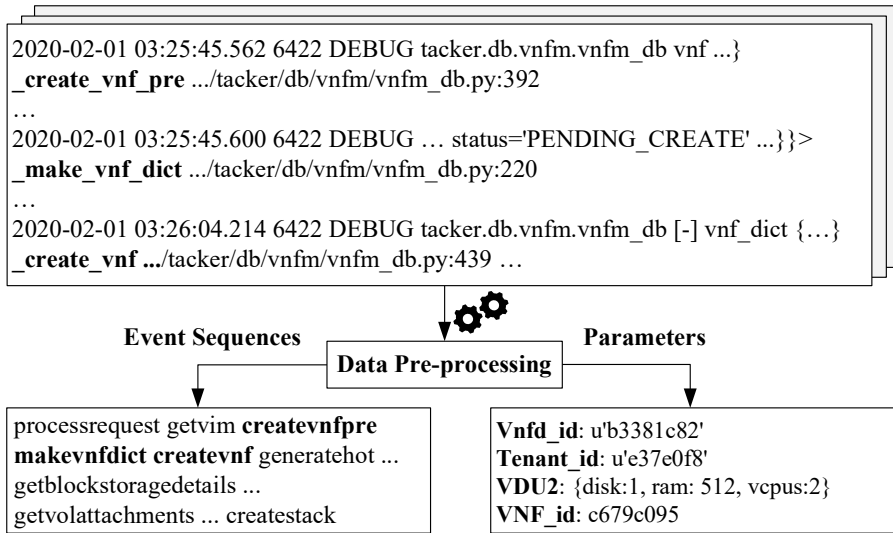


Figure 11: Example Tacker log entries and the output of data pre-processing (event sequences and corresponding parameters)

step also outputs the required parameters to generate the user-level specifications (e.g., as TOSCA templates [60]) to further support resource-level inconsistency detection (as discussed in Chapter 4.6.4). Finally, the output of our data preparation step is an NFV data corpus ready to be fed into the training modules (which addresses the third challenge).

#### 4.4.2 Event Embedding Generation

After building the data corpus, *CiT* generates event embedding similarly as generating word embedding [42] in NMT. The event embedding provides numerical feature vectors to capture the contextual meaning of events in the input corpus. *CiT* leverages the skip-gram model of *word2vec* [53] to generate event embedding, which is denoted as a multidimensional vector  $W$ . Specifically, *CiT* performs tokenization by assigning a unique index to each event, which is simply an integer  $ID(i)$  for this event. Then for each event  $i$ , the word embedding model generates the  $i^{th}$  line of  $W$ . At the end of this step, we obtain the embeddings of all events in our vocabulary, i.e., all distinct event types. We illustrate the results of our event embedding model using t-SNE [45] in Chapter 5.2.5.

**Example 6.** The embedding of the event *createvnf* (in Figure 11) is generated using *word2vec* by learning the meaning from the corpus (i.e., the input event sequences). The embedding of this event is a fixed length (e.g., 200) array which contains digitized features, e.g.,  $\langle -0.794841945, -0.719301224 - 0.445498794, -0.859628499, \dots \rangle$ . We will utilize such embeddings in the translation step.

## 4.5 Neural Machine Translation

This chapter details how CiT translates embedded event sequences between different levels of an NFV stack by leveraging neural machine translation (NMT), particularly long short-term memory (LSTM) [32]. We first present some background on LSTM and then describe our translation approach.

### 4.5.1 Background on LSTM

The long short-term memory (LSTM) [32] is an artificial recurrent neural network that supports sequence-to-sequence learning (Seq2Seq) [97], a mechanism for training machine learning models to convert text sequences from one language (e.g., English) to another (e.g., French) by capturing the meaning of those sentences using fixed-length embeddings. Particularly, the LSTM Encoder-Decoder model [49] has been shown to achieve good performance especially for long sequences [97]. Therefore, we adopt this Encoder-Decoder model for translation since the NFV event sequences are relatively long.

### 4.5.2 Event Sequence Translation

*CiT* performs event sequence translation in three main steps: choosing hyperparameters, training LSTM models, and translating event sequences using the models. First, the hyperparameters (i.e., learning parameters that play a major role in shaping and maximizing the

performance of translation model [89]) used in LSTM for natural language translation or other domains are not necessarily applicable to our context. Therefore, *CiT* evaluates different combinations of the hyperparameters that may affect the performance and accuracy of the models (see Chapter 5.2.8).

Second, to train the LSTM model, *CiT* constructs pairs of event sequences at two different levels that correspond to the same user-level operation. For instance, a sequence of normal events corresponding to a specific operation (e.g., *Create VNF*) at the VNF level is provided as the input to LSTM, with the corresponding normal sequence of events related to the same operation but at the NFVI level as the reference. Similarly, we train LSTM models for each pair of levels in the NFV stack. Finally, *CiT* applies the trained LSTM models to translate event sequences from each evaluated level to the corresponding reference level in the NFV stack in order to facilitate the next step of inconsistency detection.

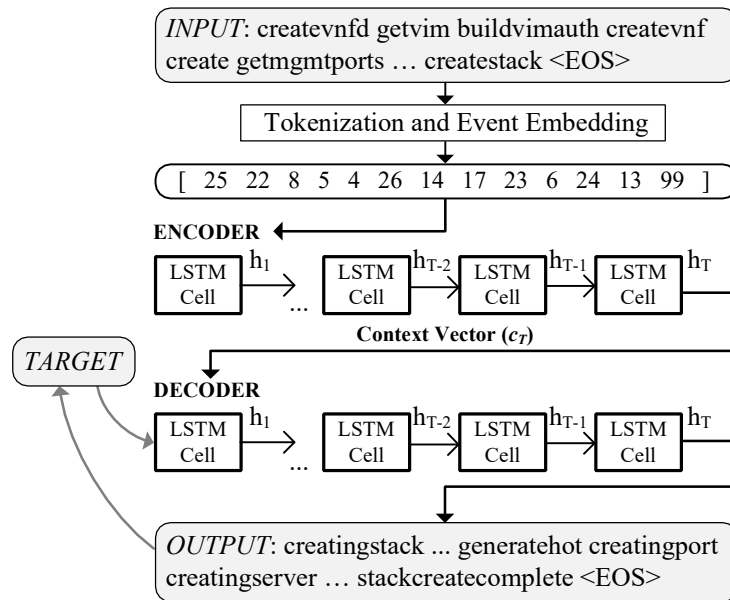


Figure 12: An example of Seq2Seq translation using the LSTM Encoder-Decoder model for an event sequence from the VNF level to the NFVI level

**Example 7.** Figure 12 shows an example of Seq2Seq translation using a trained LSTM

Encoder-Decoder model. The input event sequence (*INPUT*) from the VNF level is translated to an output event sequence (*OUTPUT*) at the NFVI level. Both *INPUT* and *OUTPUT* correspond to the same NFV user-level operation, *Create VNF*. More specifically, the event embeddings of *INPUT* are fed into the LSTM Encoder. Each LSTM *cell* in the Encoder accepts an event,  $i$ , from *INPUT* and produces a corresponding hidden state,  $h_i$ . The last hidden state of Encoder,  $h_T$ , is the sequence embedding of *INPUT* and is denoted as *context vector* ( $c_T$ ). The Decoder, which is initialized with an arbitrary start event, *TARGET*, takes  $c_T$  as input and translates the next event (e.g., *creatingstack*). The translated event is then appended to both *OUTPUT* and *TARGET*. This translation process is repeated until the Decoder generates the end-event ( $\langle EOS \rangle$ ).

## 4.6 Neural Network-based Inconsistency Detection

This chapter describes how *CiT* detects inconsistency attacks by comparing the similarities between the translated and actual event sequences using both neural network similarity learning (e.g., Siamese [55]) and similarity metric (e.g., Manhattan Distance [91]).

### 4.6.1 Background

The Siamese network [55] is a deep learning model that has been applied to find similarities between two comparable inputs. Specifically, Siamese network employs two identical deep learning models that share the same weights of parameters; where each model takes encoded inputs and generates its semantic representation as outputs. On the other hand, Manhattan distance [91] is a similarity metric that can be applied to evaluate the distance of two inputs and is mainly used with high dimensional data (such as ours).

## 4.6.2 Training

To facilitate the training of Siamese networks for detection, we prepare two kinds of training datasets: (i) the sequence of events at the same level (e.g., Tacker), and (ii) the sequence of events at two different levels (e.g., Tacker and Heat). For each pair of sequences, we also provide a ground truth similarity score based on whether the two sequences correspond to the same user-level operation; if so, a pair is called a *consistent pair*; otherwise, it is an *inconsistent pair*. The training process will vary depending on which variation, i.e.,  $CiT_{ts}$ ,  $CiT_{tm}$ , or  $CiT_s$  (as mentioned in Chapter 4.3) of our detection technique is involved. First, since  $CiT_{ts}$  translates the embedded event sequence from the evaluated level to the reference level and then compares the translated result to the embedded event sequence at the reference level, the Siamese network is trained using the first kind of dataset (i.e., event sequences at the same level). Second, no training is needed for  $CiT_{tm}$  as it compares using a similarity metric (e.g., Manhattan distance) instead of Siamese network. Third, since  $CiT_s$  directly applies Siamese network on the embedded event sequences at both the evaluated and reference levels, the Siamese network is trained using the second kind of dataset (i.e., event sequences at two different levels).

**Example 8.** Figure 13 shows the training of the Siamese network with a pair of event sequences  $S_1$  and  $S_2$ . The sequences  $S_1$  and  $S_2$  correspond to the same user-level operation, *Create VNF*, but with certain differences (highlighted). Since both event sequences are related to the same operation, this is a consistent pair and its ground truth similarity score is 1. First, the events in  $S_1$  and  $S_2$  are embedded and fed into two LSTM models (LSTM<sub>1</sub> and LSTM<sub>2</sub>), respectively. The LSTM networks then generate the event sequence embedding for  $S_1$  and  $S_2$  as hidden state vectors,  $h_T^1$  and  $h_T^2$ , respectively. As shown on the top of the figure, the similarity score is calculated as the negative exponent of the Manhattan distance calculated using  $h_T^1$  and  $h_T^2$ . The training of the Siamese network will continue until the loss function of the trained model reaches the minimum value. The parameters involved in

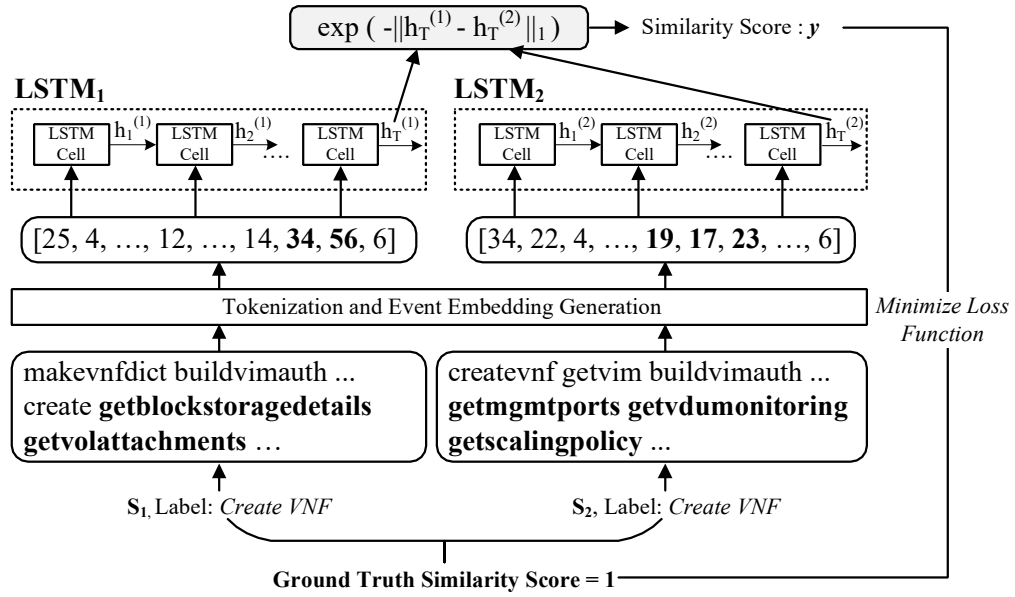


Figure 13: An example of training the Siamese Manhattan Network with NFV event sequences

Siamese training is further explained in Chapter 5.2.6.

### 4.6.3 Inconsistency Detection

CiT detects inconsistency attacks at two different levels of granularity: event-level (i.e., within the execution of one user-level operation) and workflow-level (i.e., during the execution of a sequence of user-level operations). Figure 14 demonstrates how *CiT* detects both kinds of attacks as follows.

*Event-Level Detection.* The first scenario is based on a real-world inconsistency that occurred in our NFV testbed, which is diagnosed to be related to a version mismatch between Neutron and OvS switches. In this scenario, *CiT* detects an event-level inconsistency attack (highlighted with dashed red lines in Figure 14), where there is a missing sub-sequence of events related to a user-level operation. More specifically, when the user level operation  $E_3$  is performed at the reference level (RL), a series of corresponding events denoted as  $e_3$

should be executed at the evaluated level (EL) (Step 1). However, due to the aforementioned version mismatch issue, the OvS flow rules are not updated, resulting in missing events. To detect this inconsistency, *CiT* first translates  $e_3$  (EL) to the corresponding event sequence  $E_3'$  (RL) (Step 2). When the translation module hits the unknown part of the event sequence  $e_3$  (missing events in our case), it will repeat the last known event, which can be observed as the repetitive events 89 in the translated event sequence. This inconsistency is detected by the Siamese network through comparing the translated  $E_3'$  to the actual  $E_3$  (Step 3.1).

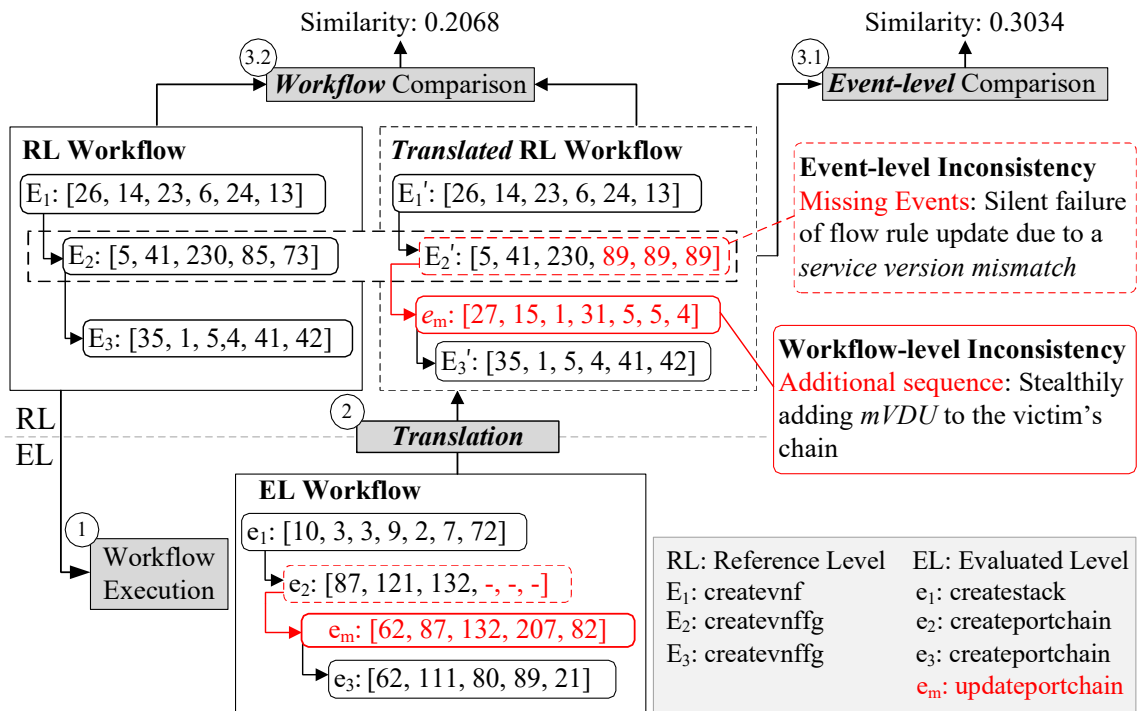


Figure 14: An example of event-level and workflow-level inconsistency detection

*Workflow-Level Detection.* In the second scenario, *CiT* detects a workflow-level inconsistency attack (highlighted with solid red lines in Figure 14), where a malicious network component (e.g., VDU) is stealthily added at the NFVI level without leaving any trace at the VNF level (similar to the attack described in our motivating example in Chapter 4.1).

In this scenario, we consider the execution of four operations ( $E_1-E_4$ ) as a workflow. During Step 1, an attacker operating at the evaluated level (EL) generates an operation  $e_m$  that has no corresponding operation at the reference level (RL). Note that the RL workflow sequence contains an embedded event sequence,  $e_m$  (shown in red), corresponding to the attack. In Step 2, the workflow sequence translated from EL to RL leads to the creation of an additional event sequence (shown in red) in the translated output, with no correspondence to the actual workflow. At Step 3.2, this inconsistency is detected through comparing the event sequences of the actual workflow and the translated workflow.

#### 4.6.4 Diff-based TOSCA Verifier

Inconsistencies may still exist after the event sequence similarity comparison as we remove the parameter values from the raw log entries as described in Chapter 4.4. As a result, we lose few implementation specific details, such as the technical specifications of a VNF, applications installed inside the VNF, etc. To compensate the loss, we utilize the parameters from service logs to generate TOSCA template that reflects the implementation details to verify the correctness of resource allocation. TOSCA template is stored in the YAML format that could be compared between the elements (i.e., requested resources, such as VDUs), e.g., `mem_size` is an attribute that denotes the memory size of a requested VDU.

---

##### **Algorithm 1:** TOSCA TEMPLATE TRANSLATOR

---

**Input:** *Services Logs, Tenant\_ID, TOSCA Template.Default*  
**Output:** *TOSCA Template.Translated*

- 1 **for** *log* in *Services Logs* **do**
- 2     *Tenant\_ID.logs* = *get(Services Logs, Tenant\_ID)*
- 3 **for** *log* in *Tenant\_ID.logs* **do**
- 4     *attribute.value, element* = *get(log)*
- 5     **search** *element* in *TOSCA Template.Default*
- 6     **add** *attribute.value* to *TOSCA Template.element*
- 7 **return** *TOSCA Template.Generated*

---

In Algorithm 1, Lines 1-2 obtain the logs that associate with the input tenant ID. The logs belong to this tenant gets separated into element, e.g., VDU, and attribute values, e.g.,



*mem\_size = 256MB* in Lines 3-4. Then the element gets identified in the input default TOSCA template and changes the attribute values into the ones obtained in Lines 5-6. Once all the logs are processed, a TOSCA template contains lower level details, which will be returned. Then Algorithm 2 takes the TOSCA user template and TOSCA generated template as inputs and generate the difference from Lines 1-3. In the end, the inconsistency between the requested resources and deployed resources will be return to the end user.

---

**Algorithm 2: TOSCA TEMPLATE COMPARISON**

---

**Input:** *TOSCA UserTemplate, TOSCA GeneratedTemplate*  
**Output:** *TOSCA diff*  
1 **d** = `diffib.Differ()` **for** *text1, text2* in *UserTemplate, GeneratedTemplate* **do**  
2     *diff* = `d.compare(text1, text2)`  
3 **return** `newline.join(diff)`

---

# Chapter 5

## Implementation and Experiments

### 5.1 Implementation of NFV Testbed

To validate our deployment model and demonstrate concrete attack scenarios, we have implemented a real NFV testbed with a telemetry NFV network service. We use OpenStack [64] as the VIM, which is considered as an essential cloud management solution by 96% of the CSPs, while more than 60% of the telecom operators are already using OpenStack for their NFV deployments [73].

OpenStack Tacker [64], an official OpenStack project for building a generic NFVM and NFVO based on ETSI MANO Architectural framework, is integrated to deploy and operate virtual network services on the VIM. We adopt the most widely used TOSCA [60] definition standards for defining network service descriptors. An ODL SDN controller is implemented to build an OpenFlow-enabled NFV system. In our implementation, Tacker uses OpenStack Heat [64] for VNF lifecycle management and user-defined VNF descriptors are uploaded to the VNFM module of Tacker through Horizon/CLI. We build our testbed on a SuperServer 6029P-WTR equipped with Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz and 128GB of RAM. Figure 15 illustrates the detailed implementation and depicts different

deployment stages as described in Chapter 2.3.

Tacker incorporates a generic driver for service function chaining, and we use the OpenDaylight SDN controller in our OpenFlow-enabled NFV system. OpenDaylight (ODL) [62] is an official open source project by the Linux Foundation. A survey conducted in May 2017 revealed that the project has over one billion users with OpenDaylight-based networks [87].

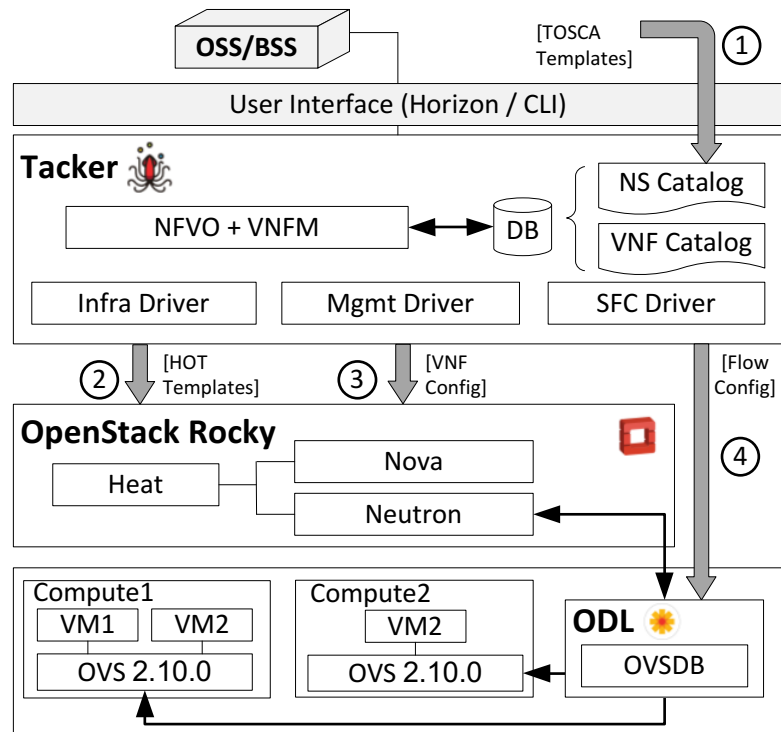


Figure 15: A real-world NFV deployment on our testbed implemented using OpenStack Tacker and ODL. The circled numbers indicate deployment stages: 1) Onboarding the NS Descriptors, 2) Deploying the VNFs, 3) Configuring the VNFs and 4) Instantiating the NS

The network service instantiation workflow involves three deployment stages: 1) onboarding VNF descriptors (VNFDs) to VNF catalog and network service descriptor (NSD) to NS catalog, 2) deploying the VNFs, 3) configuring the VNFs and 4) instantiating the network service.

### 5.1.1 Onboarding Network Service Descriptors

User-defined VNF descriptors are uploaded to the VNFM module of Tacker through Horizon/CLI. These descriptors are stored in the VNF catalog serve as VNF deployment templates. The VNFD includes technical attributes of the incorporated VDUs, such as, compute specifications (memory size, number of CPUs, image, etc.), connections points (virtual ports) and virtual links. VNFDs are validated and stored in the Tacker database for VNF instantiation.

For instance, the system specification of vFw may include the following properties which are used by the VIM during deployment: *host: VDU2; image: pfsense-fw; flavor: m1.small;*. The descriptor could also have network function configuration information, placement policies, monitoring tasks and auto-healing properties. For example, a firewall policy to allow HTTP traffic to be configured in vFw as *config rule: [name: 'allow-HTTP'; src: 'wan'; proto: 'http'; family: 'ipv4'; action: 'accept'];*.

Once VNFDs are in place, a user uploads network service descriptors to the NFVO module of OpenStack Tacker. NSDs are validated and stored in the NS catalog of NFVO. The network service descriptors mainly contain the topology description of all the network services to be instantiated in the form of forwarding graph definitions. Firstly, the VNFs belong to the network service are imported in the corresponding NSD. Then the network paths are defined by referring the connection points of the participating VNFs. The NSD template facilitates the service function chain implementation.

### 5.1.2 Instantiating the Network Service

Network service instantiation involves creating the referred VNFs followed by the deployment of service function chain. Tacker uses OpenStack Heat [65] for VNF instantiation and termination. When the *ns-create* request is submitted by a user, NFVO extracts the VNFDs and converts them to heat orchestration templates (HOTs) using the heat-translator. The

heat templates are then passed to heat-apis to orchestrate the requested VNFs. Once the VNFs are created, Tacker initiates the chain implementation through the *sfc-driver*, which is configured to use OpenDaylight (ODL) controller for managing the virtual switches. The SFC module gathers the network topology information from OpenStack Neutron [67] and generates the flow-rules to be installed. The new flow-rules are transferred to ODL which in turn updates the OpenvSwitches to route packets through VNFs.

### 5.1.3 NFV Testbed Components

This chapter introduces the various components in our NFV testbed corresponding to our NFV deployment model which is introduced in Chapter 2.3.

#### L1 Components

Figure 16 shows the VNF catalog module of Tacker’s VNFM component where the VNFDs are onboarded. Only the VNFDs which have passed the input validation will be listed as available as seen in the screenshot.

The onboarded VNFDs can then be used to deploy one or more VNFs Figure 17 shows the deployed VNFs which are in `ACTIVE` state. VNFs will go into `ERROR` state if there is any failure in the lower level deployment. For example, if there are insufficient number of `vCPUs` to create a requested VNF, the following error message will be thrown: “No valid host was found. There are not enough hosts available”.

Similar to the VNF catalog module, Tacker comprises of a VNFFG Catalog where the VNFFGDs are onboarded as shown in Figure 18. These VNFFGDs are then used to deploy VNFFGs to steer traffic through the VNFs. Figure 19 shows the VNFFG Manager component where the `ACTIVE` VNFFGs are listed.

Default • nfvds1 | nfv\_user

NFV / VNF Management /

### VNF Catalog

Filter  [+ Onboard VNF](#) [Delete VNFs](#)

Displaying 30 Items

<input type="checkbox"/>	Name	Description	Service Types	Catalog Id
<input type="checkbox"/>	vnfd-networkonenet	Demo VNFD with custom net...	-	012cfb5-6dcb-4297-8ab7-d2c6c82bc
<input type="checkbox"/>	vnfd-vdu-fip-network	Example Floating IP - All...	-	028aa466-b5d4-42ff-b81b-44bc3f1545e5
<input type="checkbox"/>	vnfd-scaling	sample-tosca-vnfd-scaling	-	03650d77-9777-4b3a-bf62-36e5ac5079c8
<input type="checkbox"/>	vnfd-scaling-flavor	sample-tosca-vnfd-scaling	-	0897c617-8728-4127-946a-1d7829f02bc6
<input type="checkbox"/>	vnfd-basic-single-vdu-threecps	Telemetry VNFD	-	0a611dbf-32a7-405a-e17f-372c6585f0b9
<input type="checkbox"/>	vnfd-alarm-espawn-flavor	Alarm VNFD	-	2fcd020-3719-429a-8c08-4cdfd50b13f0
<input type="checkbox"/>	vnfd-networkflavor	Demo VNFD with custom net...	-	3166fab6-3a99-42b3-99b1-994ab8288da6
<input type="checkbox"/>	vnfd-basic-three-vdus	Telemetry VNFD	-	360250bc-ccc64-4361-a8b6-e11dcf27947c
<input type="checkbox"/>	vnfd-basic-single-vdu-twocps	Telemetry VNFD	-	3c482c68-d2c9-45c3-b920-ae901607adf1
<input type="checkbox"/>	vnfd-basic-single-vdu-threecps-flavor	Telemetry VNFD	-	4517cf79-5e8a-4bea-9d0a-c7ff6f7055a2
<input type="checkbox"/>	vnfd-basic-single-vdu-twocps-flavor	Telemetry VNFD	-	4927cf80-9ccd-46fa-a3b1-0efc39d13fdf
<input type="checkbox"/>	vnfd-vdu-fip-network-flavor	Example Floating IP - All...	-	505eed9a-9b34-4027-b86c-03eed55141c8b
<input type="checkbox"/>	vnfd-basic-two-vdus	Telemetry VNFD	-	509f6db3-f9dc-4c7e-a9e4-9629304c8dae

Figure 16: VNF catalog in Tacker Horizon showing a list of onboarded VNFDs

Default • nfvds1 | nfv\_user

NFV / VNF Management /

### VNF Manager

Filter  [+ Deploy VNF](#) [Terminate VNFs](#)

Displaying 200 Items | [Next »](#)

<input type="checkbox"/>	VNF Name	Description	Deployed Services	VIM	Status	Error Reason
<input type="checkbox"/>	nfvds10-VNF204	vnfd4		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds14-VNF48	vnfd8		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds16-VNF67	vnfd7		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds15-VNF1510	vnfd10		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds12-VNF29	vnfd9		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds16-VNF169	vnfd9		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds9-VNF96	vnfd6		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds4-VNF49	vnfd9		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds5-VNF58	vnfd8		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds15-VNF151	vnfd1		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds11-VNF116	vnfd6		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds4-VNF41	vnfd1		default-vim	ACTIVE	-
<input type="checkbox"/>	nfvds9-VNF910	vnfd10		default-vim	ACTIVE	-

Figure 17: VNFM module in Tacker Horizon showing a list of deployed VNFs

Default • nfvds1 | nfv\_user

NFV / NFV Orchestration /

### VNFFG Catalog

Filter  [+ Onboard VNFFG](#) [Delete VNFFGs](#)

Displaying 189 items

<input type="checkbox"/>	Name	Description	Catalog Id
<input type="checkbox"/>	nfvds3-VNFFGD38	VNFFGD8	01baaa7c-8ff5-4aa5-b781-4f24cc5a9984
<input type="checkbox"/>	nfvds17-VNFFGD177	VNFFGD7	03cfe27f-38a0-4414-8669-bc2952dbc3
<input type="checkbox"/>	nfvds4-VNFFGD47	VNFFGD7	046fd9fa-55ce-426e-9782-1792269aee7f
<input type="checkbox"/>	nfvds10-VNFFGD101	VNFFGD1	070aec03-e46f-4dad-af68-b3ca523bf49b
<input type="checkbox"/>	nfvds12-VNFFGD123	VNFFGD3	073b45dd-ef0d-48c7-b9f3-6857f491079c
<input type="checkbox"/>	nfvds8-VNFFGD89	VNFFGD9	08247003-7061-4f95-a376-3db78dcaed3
<input type="checkbox"/>	nfvds14-VNFFGD143	VNFFGD3	0845e3a9-2809-4923-a4ad-dffbacc607ed
<input type="checkbox"/>	nfvds7-VNFFGD74	VNFFGD4	091cdd97-487d-48b9-86ef-48d0a5fa49eb
<input type="checkbox"/>	nfvds5-VNFFGD510	VNFFGD10	09fe3973-6b06-4858-b1ee-5c15f5019565
<input type="checkbox"/>	nfvds2-VNFFGD22	VNFFGD2	0b504abc-3b91-4037-a7be-04807c5e75b5
<input type="checkbox"/>	nfvds18-VNFFGD184	VNFFGD4	0bb2a745-4642-43ba-b238-a993fce6f319
<input type="checkbox"/>	nfvds18-VNFFGD185	VNFFGD5	0e2381fc-2ee7-4a0e-8a24-243c364fde73
<input type="checkbox"/>	nfvds3-VNFFGD31	VNFFGD1	0ffed176-7ed1-479a-a006-686f8e5dedab

Figure 18: VNFFG catalog in Tacker Horizon showing a list of onboarded VNFFGDs

Default • nfvds1 | nfv\_user

NFV / NFV Orchestration /

### VNFFG Manager

Filter  [+ Deploy VNFFG](#) [Terminate VNFFGs](#)

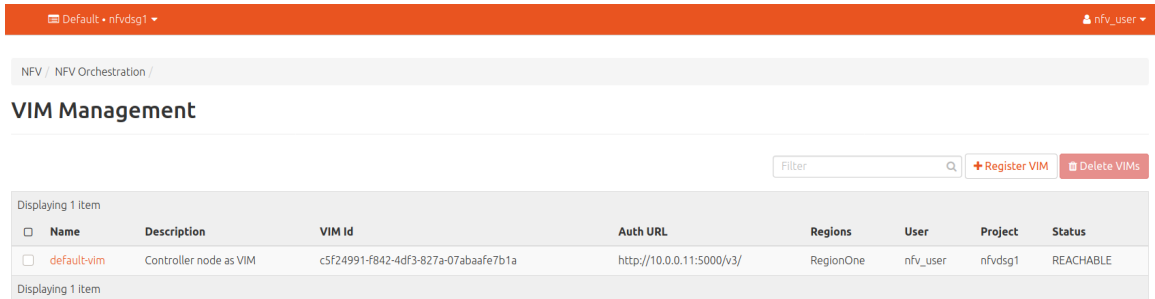
Displaying 188 items | Next »

<input type="checkbox"/>	VNFFG Name	Description	Status
<input type="checkbox"/>	nfvds11-VNFFG111	VNFFGD1	ACTIVE
<input type="checkbox"/>	nfvds10-VNFFG109	VNFFGD9	ACTIVE
<input type="checkbox"/>	nfvds1-vnffg8	VNFFGD8	ACTIVE
<input type="checkbox"/>	nfvds3-VNFFG31	VNFFGD1	ACTIVE
<input type="checkbox"/>	nfvds1-vnffg3	VNFFGD3	ACTIVE
<input type="checkbox"/>	nfvds8-VNFFG83	VNFFGD3	ACTIVE
<input type="checkbox"/>	nfvds3-VNFFG32	VNFFGD2	ACTIVE
<input type="checkbox"/>	nfvds13-VNFFG132	VNFFGD2	ACTIVE
<input type="checkbox"/>	nfvds12-VNFFG127	VNFFGD7	ACTIVE
<input type="checkbox"/>	nfvds6-VNFFG610	VNFFGD10	ACTIVE
<input type="checkbox"/>	nfvds14-VNFFG145	VNFFGD5	ACTIVE
<input type="checkbox"/>	nfvds9-VNFFG96	VNFFGD6	ACTIVE
<input type="checkbox"/>	nfvds15-VNFFG159	VNFFGD9	ACTIVE

Figure 19: VNFFG manager module in Tacker Horizon showing a list of deployed VNFFGs

## L2 Components

Figure 20 show the screenshot of NFVO component where OpenStack has been registered as a VIM (L2 component) which provides the virtual resources to build NFV systems.

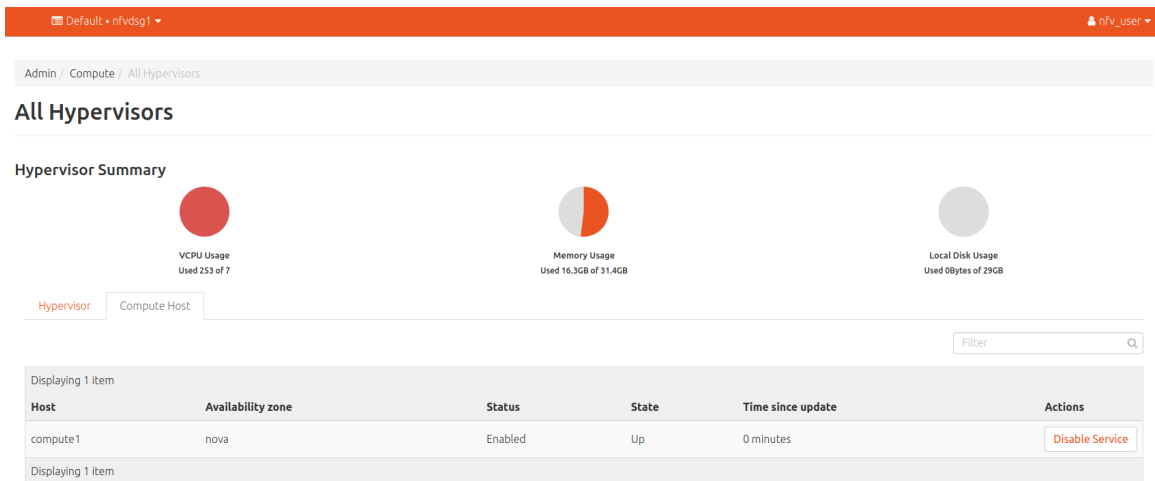


The screenshot shows the 'VIM Management' page in the NFVO interface. It features a search bar, '+ Register VIM' and 'Delete VIMs' buttons, and a table with the following data:

Name	Description	VIM Id	Auth URL	Regions	User	Project	Status
default-vim	Controller node as VIM	c5f24991-f842-4df3-827a-07abaafe7b1a	http://10.0.0.11:5000/v3/	RegionOne	nfv_user	nfvdsg1	REACHABLE

Figure 20: OpenStack VIM

Figure 21 highlights the OpenStack system overview, whereas Figure 22 gives detailed view of all the available virtual resources. OpenStack includes a stack of compute, storage, networking and orchestration services to manage the virtual resources in order to facilitate network functions virtualization.



The screenshot shows the 'All Hypervisors' page in the OpenStack interface. It includes a 'Hypervisor Summary' section with three circular gauges for VCPU Usage (Used 253 of 7), Memory Usage (Used 16.3GB of 31.4GB), and Local Disk Usage (Used 0Bytes of 29GB). Below this is a table of hypervisors:

Host	Availability zone	Status	State	Time since update	Actions
compute1	nova	Enabled	Up	0 minutes	Disable Service

Figure 21: OpenStack system overview



The screenshot shows the OpenStack Admin interface for 'Default • nfvds1'. The breadcrumb is 'Admin / System / Defaults'. The page title is 'Defaults'. There are three tabs: 'Compute Quotas' (selected), 'Volume Quotas', and 'Network Quotas'. A search filter and an 'Update Defaults' button are present. The table below lists 12 quota items with their names and limits.

Quota Name	Limit
Injected File Content Bytes	10240
Metadata Items	128
Server Group Members	10
Server Groups	10
RAM (MB)	51200
Key Pairs	100
Length of Injected File Path	255
Instances	10
Injected Files	5
VCPUs	20

Figure 22: OpenStack system defaults

Figure 23 lists all the OpenStack services which are deployed in our NFV testbed.

```
osbash@controller:~$ openstack service list
+-----+-----+-----+
| ID                | Name      | Type          |
+-----+-----+-----+
| 0a0d40f075ac451fb7dd2f1a9f863aea | nova      | compute      |
| 531651b79b3649e09a49612a01a40fce | barbican  | key-manager  |
| 56098791f5e14981ad4f3a6f682d275a | cinderv2  | volumev2     |
| 6acce1f92d1748fbb556892572d4d9c3 | tacker    | nfv-orchestration |
| 70c8706e320143298a8076fc5d804b27 | glance    | image        |
| 7fc9f957e0cf4ee7ae9d74ae41fc046f | keystone  | identity     |
| 8a41f0a70a5048c3a965aad75c1af6aa | mistral   | workflowv2   |
| 97df403900294a43a6610ae194d6e349 | heat-cfn  | cloudformation |
| 9f0c866e883d4d4a83c7f06b19651707 | neutron   | network      |
| c247845dbafd4d1ea14a361f2b8a8fe0 | cinderv3  | volumev3     |
| cc3cd5a33b34409a9f4d2c5f57501a8f | heat      | orchestration |
| f5c5ce782744417e90091dab906cb62e | placement | placement    |
+-----+-----+-----+
```

Figure 23: A list of all the OpenStack services in the NFV testbed

## L3 Components

As we have mentioned earlier, OpenFlow rules are considered as the most significant entity in L3, since they control the traffic which is steered through the VNFs. Figure 24 shows an excerpt of OpenFlow rules from the virtual switches deployed in our NFV testbed.

```
oobash@compute1:~$ sudo ovs-ofctl dump-flows br-lnt
cookie=xb6a05cf664fde7f0, duration=424.1035, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo01cd2344-43",nw_src=192.2.101.16,nw_dst=202.157.3.34,tp_dst=5844 actions=group:11
cookie=xb6a05cf664fde7f0, duration=420.4165, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo01cd2344-43",nw_src=192.2.101.16,nw_dst=127.245.181.117,tp_dst=6299 actions=group:13
cookie=xb6a05cf664fde7f0, duration=4412.1965, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo01cd2344-43",nw_src=192.2.101.16,nw_dst=93.139.247.197,tp_dst=5974 actions=group:13
cookie=xb6a05cf664fde7f0, duration=4391.0185, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo08a39266-1a",nw_src=192.5.101.18,nw_dst=210.200.188.223,tp_dst=5877 actions=group:40
cookie=xb6a05cf664fde7f0, duration=4387.2385, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo08a39266-1a",nw_src=192.5.101.18,nw_dst=210.200.188.223,tp_dst=5876 actions=group:40
cookie=xb6a05cf664fde7f0, duration=392.5435, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo49272bd1-bd",nw_src=192.6.102.11,nw_dst=136.54.231.72,tp_dst=515 actions=NORMAL
cookie=xb6a05cf664fde7f0, duration=4305.9185, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo55b87c98-ee",nw_src=192.1.101.14,nw_dst=222.161.70.25,tp_dst=971 actions=group:9
cookie=xb6a05cf664fde7f0, duration=4265.6025, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo6f379e61-b2",nw_src=192.1.101.14,nw_dst=222.161.70.25,tp_dst=971 actions=group:7
cookie=xb6a05cf664fde7f0, duration=4258.8345, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo6f379e61-b2",nw_src=192.1.101.14,nw_dst=56.143.220.171,tp_dst=863 actions=group:6
cookie=xb6a05cf664fde7f0, duration=4231.3095, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo544d5966-d9",nw_src=192.6.102.11,nw_dst=136.54.231.72,tp_dst=515 actions=group:55
cookie=xb6a05cf664fde7f0, duration=4213.4385, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo544d5966-d9",nw_src=192.6.102.11,nw_dst=199.65.29.159,tp_dst=55812 actions=group:55
cookie=xb6a05cf664fde7f0, duration=4211.9235, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo544d5966-d9",nw_src=192.6.102.11,nw_dst=128.197.179.43,tp_dst=91 actions=group:56
cookie=xb6a05cf664fde7f0, duration=4205.6525, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo544d5966-d9",nw_src=192.6.102.11,nw_dst=128.197.179.43,tp_dst=43940 actions=group:56
cookie=xb6a05cf664fde7f0, duration=4159.8845, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo0cbfa2669-1f",nw_src=192.3.101.14,nw_dst=73.14.135.284,tp_dst=329 actions=NORMAL
cookie=xb6a05cf664fde7f0, duration=4142.8435, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo0cbfa2669-1f",nw_src=192.3.101.14,nw_dst=137.159.7.39,tp_dst=383 actions=group:29
cookie=xb6a05cf664fde7f0, duration=4132.9035, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo0cbfa2669-1f",nw_src=192.3.101.14,nw_dst=44.254.80.241,tp_dst=12826 actions=group:25
cookie=xb6a05cf664fde7f0, duration=4131.6405, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo0cbfa2669-1f",nw_src=192.3.101.14,nw_dst=90.146.80.160,tp_dst=139 actions=group:38
cookie=xb6a05cf664fde7f0, duration=4091.1015, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo7639bbb-98",nw_src=192.4.102.12,nw_dst=213.182.148.224,tp_dst=64454 actions=group:31
cookie=xb6a05cf664fde7f0, duration=4084.8665, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo7639bbb-98",nw_src=192.4.102.12,nw_dst=247.133.211.252,tp_dst=35448 actions=group:32
cookie=xb6a05cf664fde7f0, duration=4083.4755, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo7639bbb-98",nw_src=192.4.102.12,nw_dst=49.75.177.141,tp_dst=699 actions=group:32
cookie=xb6a05cf664fde7f0, duration=4076.8935, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo7639bbb-98",nw_src=192.4.102.12,nw_dst=96.183.217.125,tp_dst=389 actions=group:32
cookie=xb6a05cf664fde7f0, duration=4070.2525, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo7639bbb-98",nw_src=192.4.102.12,nw_dst=164.175.2.178,tp_dst=155 actions=group:31
cookie=xb6a05cf664fde7f0, duration=4063.8855, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo7639bbb-98",nw_src=192.4.102.12,nw_dst=164.175.2.178,tp_dst=24172 actions=group:31
cookie=xb6a05cf664fde7f0, duration=4003.8945, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo25d3c19-fc",nw_src=192.5.102.16,nw_dst=191.97.3.127,tp_dst=727 actions=group:44
cookie=xb6a05cf664fde7f0, duration=3999.4455, table=0, n_packets=0, n_bytes=0, priority=30,tcp,in_port="qvo25d3c19-fc",nw_src=192.5.102.16,nw_dst=124.231.181.20,tp_dst=293 actions=NORMAL
```

Figure 24: OpenFlow Rules corresponding to the deployed VNFFGs

## 5.2 Experiments

In this chapter, we evaluate the performance of *CiT* in terms of accuracy, efficiency, robustness, and usability using both real data (obtained from a real cloud hosted at one of the largest telecommunications vendors) and synthetic data (collected from our NFV testbed). First, Chapters 5.2.1 and 5.2.2 provide the implementation details and dataset description, respectively. Chapter 5.2.4 then examines the impact of data pre-processing on out-of-vocabulary events. Finally, Chapters 5.2.6 through 5.2.9 study the accuracy, robustness, hyperparameter selection, and efficiency and scalability of *CiT*.

### 5.2.1 Implementation Details and Experimental Settings

In both the translation and inconsistency detection modules of *CiT*, we leverage the *word2vec* [53] model from *Genism* [88] to learn event embeddings from the corpus. The embedding vector is then fed into the embedding layer implemented based on *Keras* [37]. The deep

Table 2: Dataset statistics (the gray shaded datasets are processed real data)

Dataset	Training			Validation			Testing			Total		
	Con.	Incon.	Total	Con.	Incon.	Total	Con.	Incon.	Total	Con.	Incon.	Total
D1: Tacker-SFC	16,023	15,628	31,651	1,960	1,996	3,956	1,968	198	2,166	19,951	17,822	37,773
D2: Tacker-Heat	65,223	64,956	130,179	8,090	8,182	16,272	8,416	785	9,201	81,729	73,923	155,652
D3: Heat-Nova	84,952	77,415	162,367	10,660	9,636	20,296	10,574	972	11,546	106,186	88,023	194,209
D4: Heat-Neutron	65,343	62,258	127,601	8,119	7,831	15,950	8,092	785	8,877	8,1554	70,874	152,428
D5: Neutron-OvS	14,022	12,344	26,366	1,674	1,622	3,296	1,742	155	1,897	17,438	14,121	31,559
D6: Heat-Nova	64,423	37,542	101,965	8,073	4,673	12,746	8,080	466	8,546	80,576	42,681	123,257
D7: Heat-Neutron	76,209	39,992	116,201	9,578	4,947	14,525	9,491	503	9,994	95,278	45,442	140,720
D8: Neutron-OvS	187,635	67,036	254,671	23,574	8,260	31,834	23,449	838	24,287	234,658	76,134	310,792
D9: Multilevel	56,758	54,625	111,383	7,078	6,845	13,923	7,061	686	7,747	70,897	62,156	133,053
D10: Multilevel	85,245	63,315	148,560	10,671	7,899	18,570	10,520	805	11,325	106,436	72,019	178,455
<b>Total</b>	715,833	495,111	1,210,944	89,477	61,891	151,368	89,393	6,193	95,586	894,703	563,195	1,457,898

Table 3: Statistics of the original real data (from May 2017 to March 2020)

Size	# of services	Duration	Heat/node (entries)	Nova/node (entries)	Neutron/node (entries)	OvS/node (entries)	Total # of event types
47.5G	10	3 years	951,053	1,977,847	3,957,313	2,950,495,169	164

learning layer, e.g., LSTM, is implemented based on the *Keras.layers* library to learn the event sequence embeddings. For comparison, we have also implemented GRU and RNN. The implementation of the evaluation metrics, e.g., Loss and AUC, is based on *scikit-learn* [92], a well-known ML library. The implementation of data preparation is based on *pandas* [81], a data analysis library. All the modules of *CiT* are developed in Python 3.7.4.

In order to test and evaluate *CiT*, we have additionally implemented an NFV testbed. OpenStack [64] is used as the VIM component, which manages the virtual infrastructure. OpenStack Tacker [98], an official OpenStack project for building generic VNFM and NFVO based on the ETSI MANO architectural framework, is used to deploy virtual network services on the VIM. All the experiments are performed on a SuperServer6029P-WTR running the Ubuntu 18.04 operating system equipped with Intel(R) Xeon(R) Bronze 3104CPU @ 1.70GHz and 128GB of RAM *without* GPUs.

## 5.2.2 Datasets

Table 2 summarizes all the datasets used in the evaluation of *CiT*. In total, we obtain 26,356 unique event sequences and generate 894,703 *consistent* pairs (i.e., two event sequences corresponding to the same user-level operation) and 563,195 *inconsistent* pairs. The datasets follow the dependencies of services in the NFV deployment model described in Chapter 2.3.

### Real-World Data

We have collected around three years of OpenStack logs from a real cloud hosted at a major telecommunications vendor with hundreds of users. Table 3 shows some statistics of the original data, which we have processed to obtain the datasets D6, D7, D8, and D10 shown in Table 2 following the approach described in Chapter 4.6.2. In doing so, we processed 47.5G of raw data and obtained 164 event types from four different services, i.e., Heat, Nova, Neutron, and OvS. To obtain realistic testing datasets, the inconsistent pairs in testing datasets are generated based on real-world bug patterns (detailed in Table 5, where the inconsistency is caused by OpenStack implementation bugs) and denied event sequences in the real data (where the inconsistency is caused by a violation). Among all the processed raw data, 2~8% of the data corresponds to denied user-level operations in different services. Therefore, we inject *inconsistent* pairs following the similar percentage. We split training, validation, and testing datasets into disjoint event sequences, so we can evaluate the ability of *CiT* towards handling unseen event sequences. We discuss some the challenges encountered while extracting the event sequences from individual services and creating pairwise event sequences in Chapter 5.2.3.

## Data Generation in the NFV Testbed

We have implemented an NFV testbed to collect datasets from the NFV stack, including from the Orchestration Level (L1) which is not present in our real data. In Table 2, datasets D1 through D5, and D9 are obtained from the NFV testbed. We used Python scripts to automatically generate TOSCA templates in order to deploy NFV entities, such as VNFs and VNFFGs. In total we have deployed 31 types of VNFs (e.g., with auto-scaling policies, dedicated subnet, floating IPs, etc.), and 7 variations of VNFFGs in order to create sufficient diversity in the corresponding event sequences. We have also randomized a few important parameters in the template description, such as 1) the number of virtual network ports per VNF, 2) the number of deployment units per VNF, 3) the node Flavor specification for each VDU, 4) the number of VNFs for each Network Forwarding 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.

### 5.2.3 Challenges in Processing the Real-World Data

We briefly describe the challenges encountered while extracting the event sequences from individual services and forming pairwise event sequences.

**Obtaining the Event Vocabularies.** Real-world data contains more diverse event vocabularies than the data we obtained from our testbed as it was generated by hundreds of real users. To address this issue, we first went through the real-world data to understand the structure of the logs for each service, e.g., *heat-engine*, then we developed regular expression to extract the event vocabularies. Data entries that do not yield any event were output for further evaluation through constructing new regular expressions to extract the missing event types. After several rounds of evaluation, we obtained a complete collection of event vocabulary from the real-world data.

**Generating Event Sequences.** Unlike the logs from our testbed, real data contains mixed user-level operations that require proper separation before they can be used to extract event sequences. To address this issue, we utilized four IDs, *request\_id*, *tenant\_id*, *domain\_id*, and *stack\_id* (stack means a group of logs that perform similar user-level operations), to group data into each user-level operation. Then we extracted the events and combined them based on time stamps to form an event sequence. User-level operation became the label for this group of data.

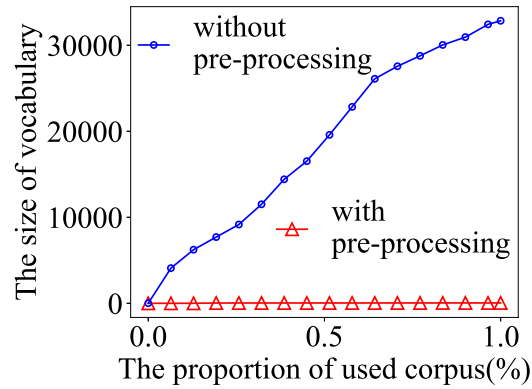
**Generating Datasets with Ground Truth.** As described in Chapter 4.6.2, in the dataset generation, two event sequences corresponding to the same user-level operation would be assigned with the *consistent* label, and vice versa. Unfortunately, the user-level operation is missing from the real data we obtained. To address this issue, we applied the model trained based on the data collected from our testbed to predict the user-level operation for the real data. All the predicted labels are then validated by two domain experts to ensure the correctness of the labeling.

#### 5.2.4 Evaluation on Out-of-Vocabulary (OOV) Events

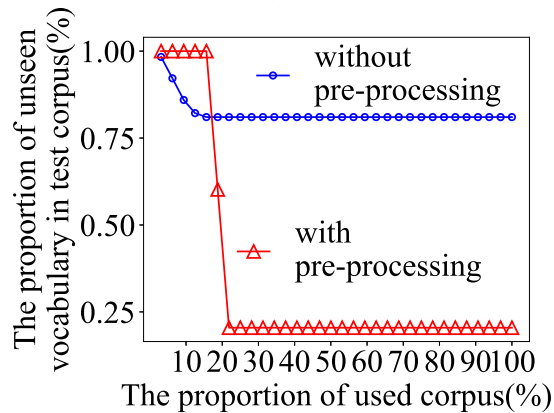
As we apply data pre-processing to address the OOV challenge (described in Chapter 4.4), we evaluate its impact in two aspects, i.e., the size of vocabulary (all distinct events form a vocabulary) and the proportion of unseen vocabulary in a new corpus, shown in Figure 25a and Figure 25b. Both experiments are performed based on the real-world data.

We count the vocabulary size in terms of the percentage of the processed corpus (data entries). The red line (with triangle marker) and the blue line (with cycle marker) and show the growth of vocabulary size with or without pre-processing. This experiment is conducted based on 951,053 Heat service log entries for three years. Since pre-processing remove non-essential implementation details and parameters, as shown in Figure 25a, when pre-processing is not applied to the dataset, the vocabulary size grows nearly linearly (reaching

more than 30,000 when 100% of the corpus is used) in the percentage of the analyzed corpus, whereas the vocabulary size stays stable when pre-processing is applied (in total 164 event types).



(a)



(b)

Figure 25: (a) The growth of vocabulary size, and (b) the proportion of unseen event types

We divide our data into two parts, with each part containing one year of the data, to investigate the number of OOV cases, i.e., unseen vocabulary, in a later year of data. We generate vocabulary based on the data from year 2018 to 2019, and then count the percentage of unseen vocabulary that does not exist in the vocabulary in the data from year 2019 to 2020. As shown in Figure 25b, the unseen vocabulary drops to less than 20% when pre-processing is applied, whereas unseen vocabulary stays at around 80% without

pre-processing. This result shows that the pre-processing significantly increases the coverage of the vocabulary; only 20% of the data from the previous year is needed to cover most of the event vocabulary in the latter year (while 80% would be needed without pre-processing).

### 5.2.5 Event Embedding Model

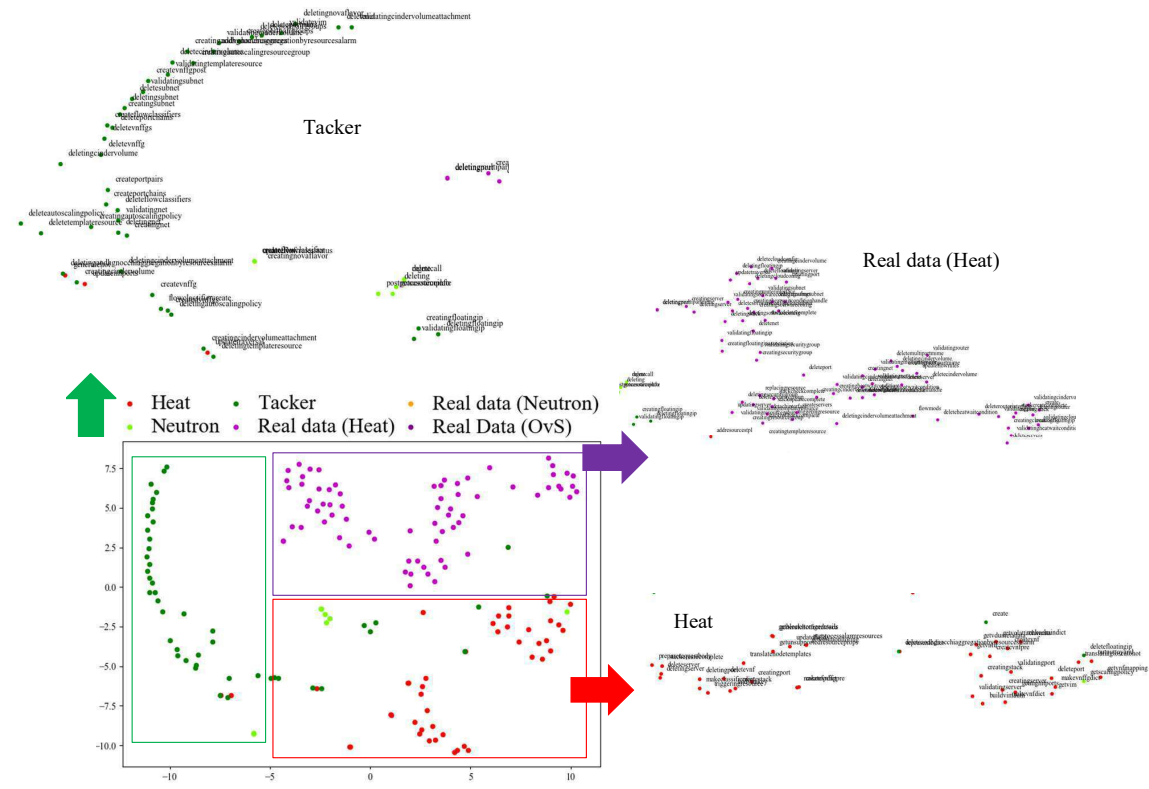


Figure 26: Visualization of NFV events with t-SNE

We utilize t-SNE clustering to visualize the embeddings of different event types in our dataset. Each token is represented as a 200-dimensional numeric vector and learned by *word2vec* using the event sequences extracted from the dataset. For visualization, t-SNE reduces the vectors to two dimensions by nearest neighbor approximation. As shown in Figure 26, all the event types are naturally separated into multiple clusters based on their



corresponding NFV service (For e.g., Heat, Tacker). Since, all the events related in service-level appear together this result shows the event embedding model learns their semantic information from the event sequences obtained from our datasets.

## 5.2.6 Inconsistency Detection Evaluation

In this chapter, we evaluate the accuracy of  $CiT$  and its three variations (as described in Chapter 4.3). First, we compare  $CiT_s$  with traditional machine learning model Support Vector Machine (SVM) [106] with the TFIDF feature set [86, 111] based on all the 10 datasets including both the real-world data and testbed data (detailed in Table 2). SVM is known as an effective method for sequence classification in the literature [106], and TFIDF is used to extract text features for anomaly detection and failure prediction [16]. Therefore, we choose SVM with TFIDF feature set for comparison with our solution. Second, we compare the three variations of  $CiT$ , i.e.,  $CiT_s$ ,  $CiT_{tm}$ , and  $CiT_{ts}$  in terms of accuracy.

### $CiT_s$ vs. SVM

Recall that  $CiT_s$  directly applies Siamese network on the embedded event sequences at two different level for inconsistency detection. The implementation of Support Vector Machine (SVM) model is from *scikit-learn*. We have evaluated the SVM model based on five kernels in which kernel *rbf* performs the best followed by kernel *poly*, and kernel *linear* ranks last. Therefore, we only present the results generated from kernel *rbf* with  $\gamma = 1.0$  and  $c = 1.0$ , which achieves the best AUC, as a comparison to study the discriminative power of TFIDF features and deep learning generated features (embedding).

**Model Training.** In this set of experiments, training, validation and testing datasets follow the statistics presented in Table 2. We use 10 datasets to train  $CiT_s$  up to 200 epochs. We implement early stopping [8] to avoid overfitting. We choose the *loss* value of the corresponding validation dataset as the performance measure with the trigger parameter set

to  $patience=3$ . While training the model,  $loss$  will be monitored as it is calculated after each epoch. If there is no improvement in the  $loss$  value for 3 epochs, the training stops. Other hyperparameters, such as the dimensions of the event embedding and the dimension of sequence embedding, are set to 256 and 200, respectively (a more detailed study of hyperparameters is presented in Chapter 5.2.8).

The training dataset for the SVM model is the same training dataset as  $CiT_s$ . The implementation of TFIDF features is from *scikit-learn*, which converts raw text inputs into a matrix of TFIDF features. In the training of this experiment, we set the dimension of the feature sets to 300 following the literature [111]. We also evaluate the performance of SVM model under small feature dimension ( $max\_features = 5$ ) and large feature dimension ( $max\_features = 1000$ ). Comparing to 300 dimensional TFIDF feature sets, small feature dimension performs worse, while large feature dimension does not show stable improvements. Thus, we only show the results with SVM model that is trained and tested on TFIDF feature sets with 300 dimensions.

**Inconsistency Detection Results for  $CiT_s$ .** We now evaluate the accuracy of  $CiT_s$  using the corresponding testing dataset of D1 to D10 that includes real bugs and denied event sequences.

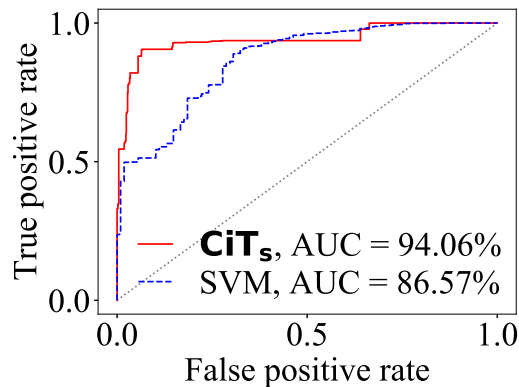


Figure 27: The ROC evaluation results of  $CiT_s$  based on D1

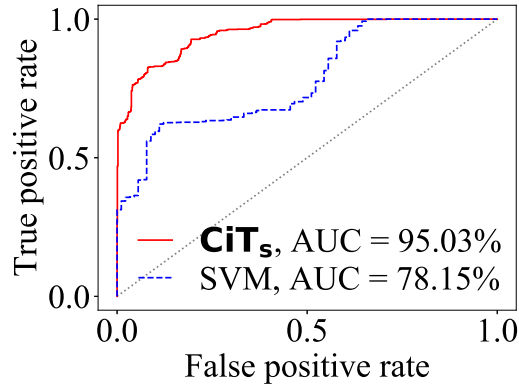


Figure 28: The ROC evaluation results of  $CiT_s$  based on D2

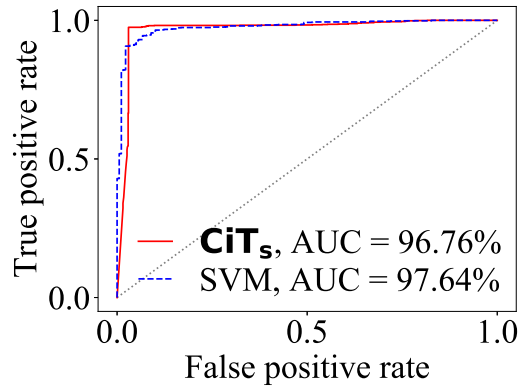


Figure 29: The ROC evaluation results of  $CiT_s$  based on D3

Under the evaluation with testbed datasets, the AUC value of  $CiT_s$  for inconsistency detection increases for all the datasets from D1 (AUC = 94.06%) to D5 (AUC = 100%). Hierarchically, datasets from the higher level services, such as, D1 (Figure 27) and D2 (Figure 28), generally consist of longer event sequences, e.g., Tacker service generates orchestration events that could be implemented with multiple lower level services; while D3 (Figure 29) to D5 (Figure 31) consist of lower level service events which generally have less events and shorter event sequences. Notably, OvS service has only one event, *flow\_mods*, to implement the requested flows from Neutron service. Thus, the inconsistency detection is relatively easy for  $CiT_s$  and SVM (both reaching  $\sim 100\%$ ). In general, the value of AUC

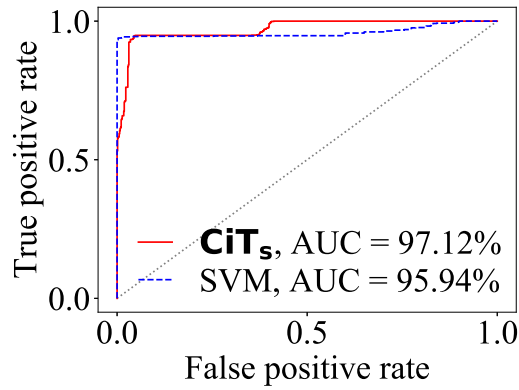


Figure 30: The ROC evaluation results of  $CiT_s$  based on D4

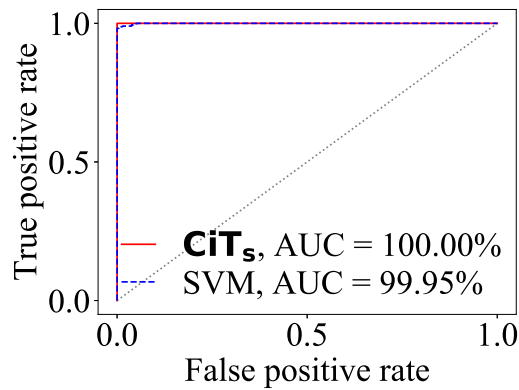


Figure 31: The ROC evaluation results of  $CiT_s$  based on D5

increases as the complexity of the datasets decreases due to event sequences from only lower level services.

Under the evaluation with real data,  $CiT_s$  performs well with D6 (Figure 32) and D8 (Figure 34), i.e., AUC increases from 98.27% to 100%, which leads to similar conclusion drawn from the testbed datasets. However, the inconsistency detection evaluation with D7 (Figure 33) results in only 85.21%. This is mainly because the complexity of this dataset is sufficiently higher due to the increased diversity of network-related events in real data, i.e., one event sequence from Heat service could be implemented in various ways based on user chosen templates. As we will show shortly, this result will be significantly improved with

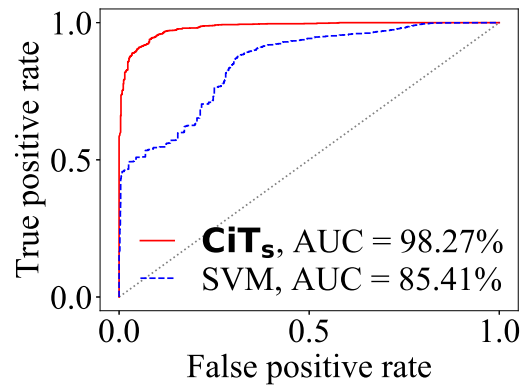


Figure 32: The ROC evaluation results of  $CiT_s$  based on D6

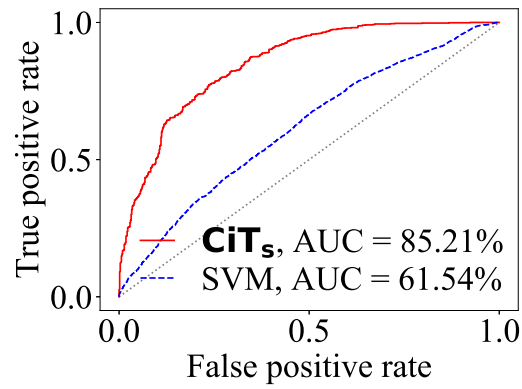


Figure 33: The ROC evaluation results of  $CiT_s$  based on D7

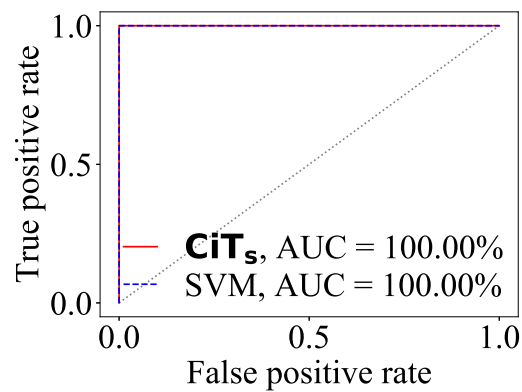


Figure 34: The ROC evaluation results of  $CiT_s$  based on D8

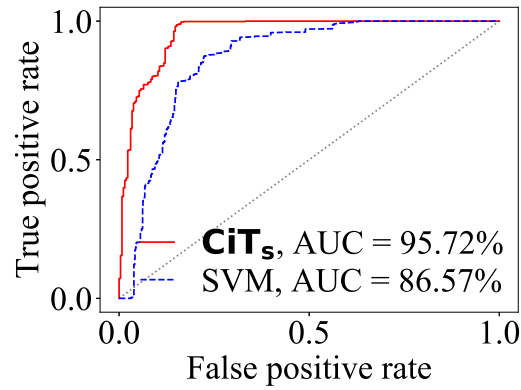


Figure 35: The ROC evaluation results of  $CiT_s$  based on D9

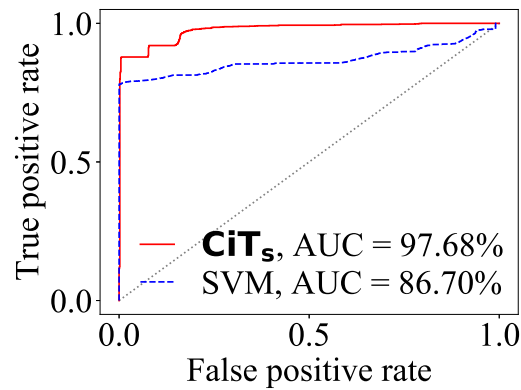


Figure 36: The ROC evaluation results of  $CiT_s$  based on D10

$CiT_{ts}$  and  $CiT_{tm}$  since translation helps to reduce the detection complexity as explained in Chapter 5.2.6. Nonetheless,  $CiT_s$  still demonstrates good inconsistency detection ability for both the testbed and real data, with  $AUC = 95.72\%$  and  $97.68\%$ , respectively. AUC is slightly lower in the testbed dataset because the long sequences from orchestration level (D1 and D2), which are missing in the real dataset, increase the complexity of the dataset.

**Comparison with SVM.** After the models of  $CiT_s$  and SVM are trained, we use the same testing datasets to evaluate both. In general, the comparison between the two models in terms of detecting inconsistencies shows similar results across different datasets. We can

observe that for all datasets, the ROC curves corresponding to  $CiT_s$  are closer to the left-hand and top border than SVM models, which indicates our model generally has better accuracy. The majority of our models yield an ROC-AUC value higher than 94%, while SVM stays around or less than 85% for six of the 10 datasets (D1, D2, D6, D7, D9, and D10). Comparing to SVM, the improvement of  $CiT_s$  is more significant when tackling the datasets with longer or more diversified event sequences. This observation confirms that the embedding and deep learning algorithms are more capable of handling complicated event sequences.

### **$CiT_s$ vs. $CiT_{tm}$ vs. $CiT_{ts}$**

In this set of experiments, we compare the three variations of  $CiT$  to evaluate the improvement of accuracy (AUC) introduced by the additional translation capability of  $CiT_{tm}$  and  $CiT_{ts}$ . Recall that  $CiT_{ts}$  applies LSTM to translate event sequences before applying Siamese network on the embedded event sequences at the same level for inconsistency detection, whereas  $CiT_{tm}$  translate event sequences but then performs inconsistency detection based on the Manhattan distance, as described in Chapter 4.6.2. For these experiments, we use the datasets (D1, D2, D7, and D9) for which  $CiT_s$  achieves relatively lower AUC (<96%).

**The Comparison of Inconsistency Detection Results.** We present the ROC curves for the inconsistency detection based on the aforementioned four datasets for all three variations of  $CiT$ . We can observe that  $CiT_{tm}$ , which performs inconsistency detection based on Manhattan distance, already achieves satisfying results (AUC>94.51%). Moreover,  $CiT_{ts}$  achieves AUC =  $\sim$ 100% in three datasets (D1, D2 and D9) shown in Figure 37, 38, and 40. Notably,  $CiT_s$  only achieves AUC = 85.21% in D7 (shown in Figure 39), whereas, with the help of translation,  $CiT_{tm}$  and  $CiT_{ts}$  achieve AUC = 94.51% and 96.03%, respectively. These results show that the translation module can significantly improve the accuracy of

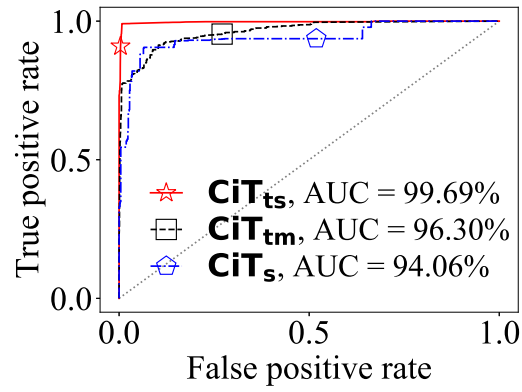


Figure 37: The ROC evaluation results of  $\text{CiT}$  based on the datasets D1

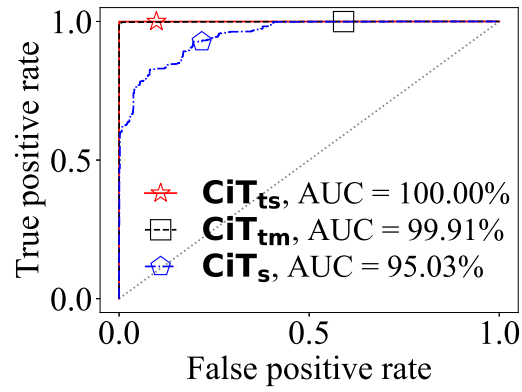


Figure 38: The ROC evaluation results of  $\text{CiT}$  based on the datasets D2

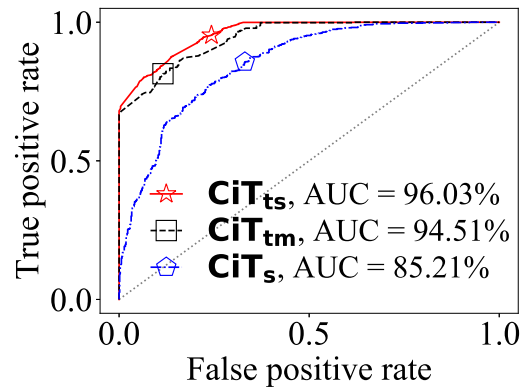


Figure 39: The ROC evaluation results of  $\text{CiT}$  based on the datasets D7



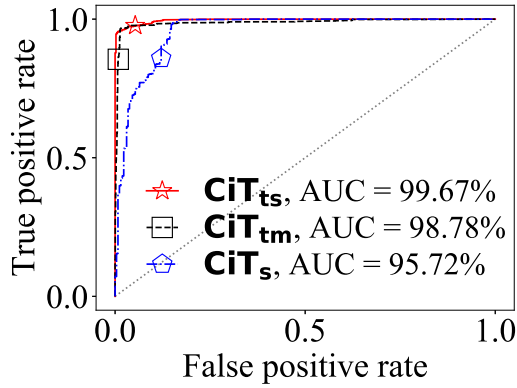


Figure 40: The ROC evaluation results of  $CiT$  based on the datasets D9

inconsistency detection. We can conclude that  $CiT_{ts}$  achieves the best results on all the datasets, followed by  $CiT_{tm}$ .

## 5.2.7 Robustness Evaluation

We conduct three case studies to evaluate  $CiT$ 's robustness.

### Training and Testing with Different Systems

We apply the inconsistency detection models trained on our testbed dataset D9 to test the real-world dataset D10. These two datasets are obtained from two very different systems, with one implemented by ourselves for experimental purposes, and the other hosted at a major telecommunications vendor with hundreds of real-world users. Therefore, this experiment evaluates the robustness of  $CiT$  when trained and tested on different NFV systems.

**Results.** The experimental results presented in Figure 41 show that, even under this challenging scenario with two significantly different datasets (e.g., even the event types are different),  $CiT_s$  still achieves an acceptable AUC (=73.58%). Aligned with the observations in Chapter 5.2.6, Figure 41 shows that the extra translation step helps to improve the

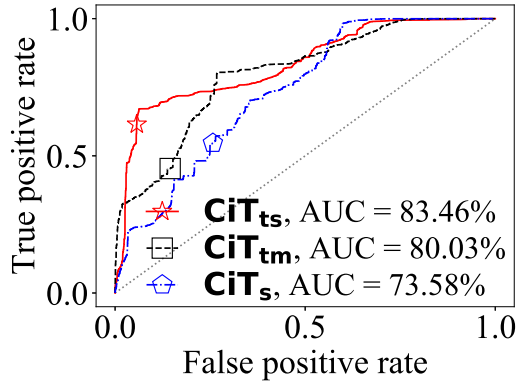


Figure 41: The ROC evaluation results of  $CiT$  based on D10 tested on D9

AUC to 80.03% ( $CiT_{tm}$ ) and 83.46% ( $CiT_{ts}$ ), respectively. These results confirm the robustness of  $CiT$  as well as the feasibility of training  $CiT$  under a controlled environment and then applying it to a real world NFV system.

### Label Translation for Real-World Data

As highlighted in Chapter 5.2.3, our real-world data does not include the corresponding user-level operations, which are needed as labels to form *consistent* and *inconsistent* sequence pairs. It is infeasible to label these manually considering that there are 8,192 unique event sequences in total in the real-world data. Thus, we utilize the translation module of  $CiT$  to translate the corresponding labels (user-level operations) of event sequences from our real-world data by feeding them as testing dataset. All the translated labels are validated by two domain experts to ensure the correctness of the labeling.

**Results.** The label translation results are presented in Table 4. We only show the most commonly occurring user-level operations for each service and their corresponding accuracy of label translation. Our translation module is able to correctly label up to  $\sim 94\%$  of the events for the Heat service followed by Nova ( $\sim 88\%$ ) and Neutron ( $\sim 75\%$ ) services. Also, most of the events are labeled with 100% accuracy except a few. Particularly, the

Service (%)	Event Sequence Label (%)	Description
Heat (~94)	CreateServer (100)	Creating a VM
	CreateStack (84)	Multiple operations
	CreateCinderVolume (94)	Creating a Cinder Volume
	DeleteServer (100)	Deleting a VM
	DeleteStack (89)	Multiple operations
Neutron (~75)	CreatePort (100)	Creating a virtual port
	CreateFloatingIP (100)	Creating a floating IP
	UpdateFlowRuleStatus (70)	Updating OvS flow rules
Nova (~88)	CreateServer (100)	Creating a VM
	OSServerExternalEvents (85)	Attaching a floating IP
	OSSecurityGroupRules (100)	Assigning a security group

Table 4: Robustness evaluation of *CiT*: Sequence label translation for the real-world data. (%) indicates the percentage of correctly translated labels.

*CreateStack* event sequences of Heat service achieve a success rate of  $\sim 84\%$ . Upon investigation, we find that the real event sequences that fall under *CreateStack* category include multiple *create*, *update* and *delete* event types since a “stack” executes multiple operations at once. These event sequences are not correctly labeled by the model trained based on the testbed data, since D9 contains only *create* event type for the label *CreateStack*. Furthermore, the Neutron service achieves comparatively the lowest success rate of  $\sim 75\%$ . The reason is that the event sequences labeled with *UpdateFlowRuleStatus* in D9 contain the Tacker service events, which are not included in dataset D10. The overall results again confirm the robustness of *CiT* in applying its trained models to a different system.

### Real-World Inconsistency Detection

To investigate how *CiT* detects real world inconsistencies, we conduct the following experiment using event sequences including both real-world bugs and denied operations found in our real data. The sequences are evaluated using *CiT*, trained with *consistent* and *inconsistent* pairs from the same level (e.g., Heat-Heat).

**Results.** Table 5 summarizes the results of this case study including the detailed description of the bugs or denied operations and their corresponding similarity scores. As we can observe, majority of the sequences are assigned with a lower similarity score ( $< 0.5$ ).

#	OpenStack Implementation Error Description	Severity	Sim. Score
Bug#1527658	Block Device Mapping is Invalid	NA	0.0038
Sequence#1	ValueError: <name>: nics are required after microversion <Version #>	NA	0.0150
Bug#1653164	CinderVolume <name> Stack <name> [id] timed out	Critical	0.1791
Sequence#2	NotFound: resource with id <id> not found	NA	0.2003
Sequence#3	Resource CHECK failed	NA	0.2395
Bug#1517355	Conflict: Port <id> is still in use	High	0.2437
Sequence#4	OverQuotaClient: Quota exceeded for resources	NA	0.3367
Sequence#5	Error: Volume in use	NA	0.4380
Sequence#6	Resource CREATE failed: You are not authorized to use resource_types <name>	NA	0.7047
Bug#1833455	Forbidden: rule <event> is disallowed by policy	Medium	0.7396
Bug#1808112	Forbidden: resources. Offline rule <event> is disallowed by policy	Medium	0.7970

Table 5: Robustness evaluation of *CiT*. Case study on real-world bugs and denied operations

Meanwhile, three of these obtain a comparatively higher similarity score ( $>0.7$ ). Our investigation shows that this is mainly due to the fact that these sequences are relatively shorter (with less events) and the bugs do not introduce significant differences to the events. For example, the event sequence corresponding to *Bug#1808112* is *creatingport createport stackcreatefailed*, whereas the normal sequence is *creatingport createport stackcreatecomplete*, i.e., only one event is different, and both are relatively short sequences. To overcome such a situation, *CiT* can be extended with a *weighted* similarity score measure (e.g., [9,43]) such that more important events, e.g., *stackcreatefailed* (which indicates the requested operation has failed due to an error), will carry more weight in the calculation of the similarity score (which is considered as a future work).

## 5.2.8 Hyperparameter Selection

In this chapter, we evaluate the impact of hyperparameters involved in training *CiT*. Specifically, we first study the AUC and loss metrics in terms of the number of epochs using five datasets (D1, D2, D7, D9, and D10). We then investigate the impact of (both event and sequence) embedding dimensions and network hidden unit types based on the multilevel datasets (D9 and D10) for the three *CiT* variations.

## Number of Epochs

Figure 42 and Figure 43 show the results of AUC and loss metrics for translation training on the aforementioned five datasets, respectively. We train the translation model for 200 epochs and evaluate the model after each epoch. The curves of both AUC and loss metrics become flat after 20 epochs for 4/5 datasets (only D7 requires around 50 epochs to become stable, as the complexity of the dataset is higher). We can also observe that the early stop implemented in each model stops around 20 epochs. In summary, we conclude that 20 epochs could be enough to obtain a good model in *CiT*.

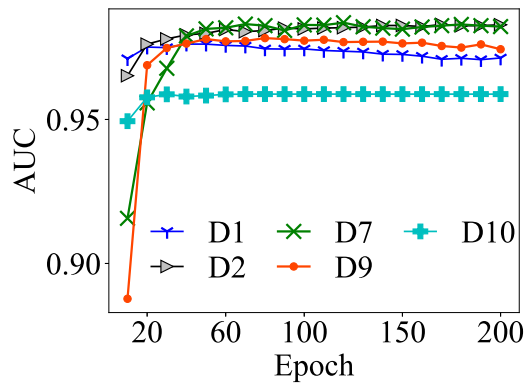


Figure 42: The impact of number of epochs on *CiT* - AUC vs. # of epochs

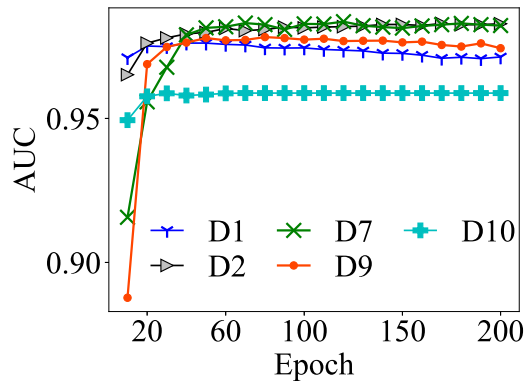


Figure 43: The impact of number of epochs on *CiT* - Loss vs. # of epochs

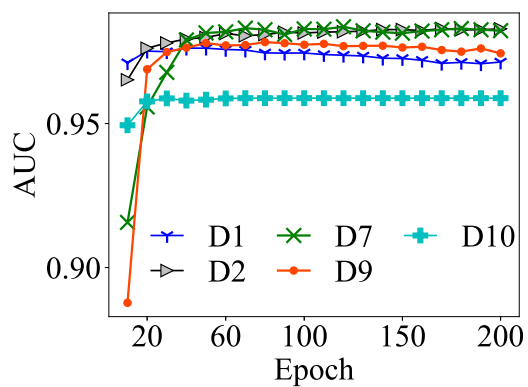


Figure 44: The impact of number of epochs on *CiT* - Accuracy vs # of Epoch

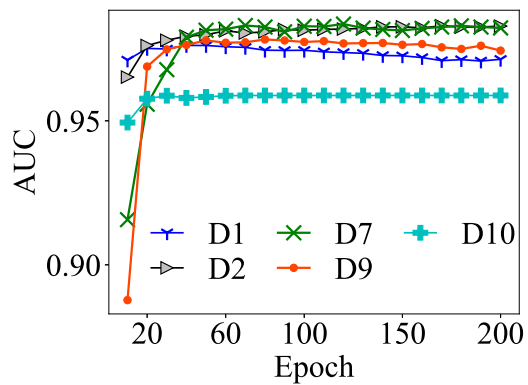


Figure 45: The impact of number of epochs on *CiT* - Precision vs # of Epoch

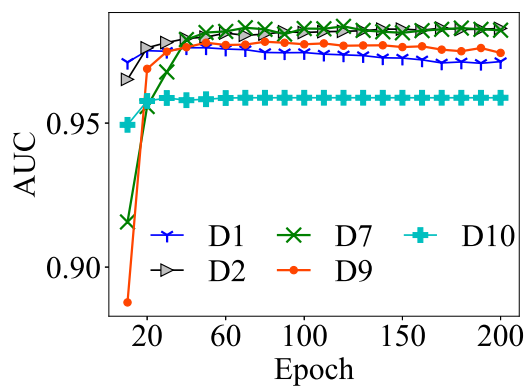


Figure 46: The impact of number of epochs on *CiT* - Recall vs # of Epoch

The results of three other metrics, accuracy, precision, and recall, for translation training process on five datasets (D1, D7, D2, D9, and D10). Align with the AUC and loss results we observed in Figure 42 and Figure 43 the metrics results become stable around 20 epochs. It also confirms that our model could be trained around 20 epochs with good performance in all the metrics in all the tested datasets (i.e., both real-world and testbed datasets).

### Data Split Ratio

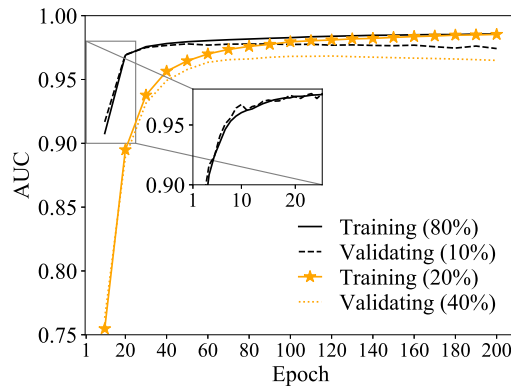


Figure 47: Evaluation of data separation ratio - AUC

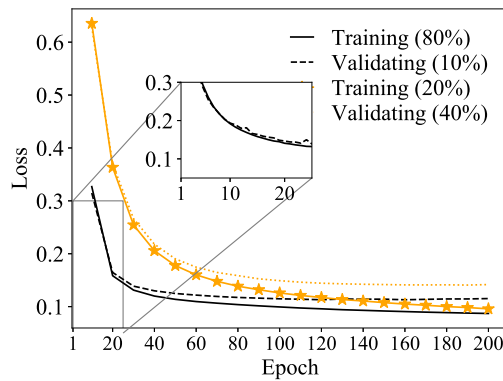


Figure 48: Evaluation of data separation ratio - Loss

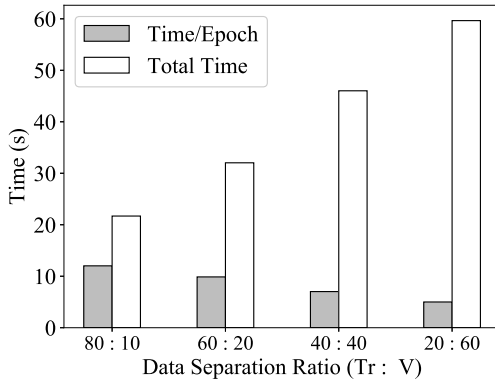


Figure 49: Evaluation of data separation ratio - Training Time

Tr: V: Te	Loss	AUC	Accuracy	Precision	Recall
0.8: 0.1: 0.1	0.108	0.975	0.994	0.811	0.801
0.6: 0.2: 0.2	0.117	0.974	0.994	0.809	0.801
0.4: 0.3: 0.3	0.121	0.969	0.994	0.801	0.792
0.2: 0.4: 0.4	0.138	0.966	0.993	0.791	0.778

Table 6: The evaluation of data separation based on five metrics

For training, we split the training, validation and testing dataset using four different split ratios in order to analyze the best split ratio. From the results, we can also infer that the split ratio of 60%, 20%, and 20% for training, validation, and testing datasets, respectively, could be considered as the optimal split ratio given the training time trade-off. Table 6 presents additional accuracy metrics to further validate the accuracy depending upon the data separation ratio.

### Event Embedding Dimensions

We study the impact of event embedding dimension ( $Em$ ) in two multi-level datasets for three  $CiT$  variations. The AUC values are presented in Table 7. We observe that, as the event embedding dimension increases, all the models yield higher AUC values. Comparing to D9, D10 achieves better results with all three variations. Overall, we can observe that a higher embedding dimension generally leads to higher computational costs and longer



training time. Considering the fact that the training is a one-time effort (until the system undergoes major changes), users could leverage such results to choose the right trade-off based on their requirements.

Em	Datasets					
	D9			D10		
	$CiT_s$	$CiT_{tm}$	$CiT_{ts}$	$CiT_s$	$CiT_{tm}$	$CiT_{ts}$
50	93.14	95.04	97.72	95.42	96.01	98.16
100	93.63	96.49	98.29	95.65	96.55	98.82
150	94.88	97.70	98.81	95.92	97.13	99.59
200	95.42	98.11	99.49	96.33	97.35	99.73
250	95.72	98.78	99.67	97.68	98.36	99.92

Table 7: AUC (%) vs. event embedding dimensions ( $U = 256$ )

### Sequence Embedding Dimension

We vary the sequence embedding size in terms of the number of LSTM network units ( $U$ ) and show the corresponding AUC results in Table 8. Similar to Table 7, the AUC values of the three variations increase with the number of sequence embedding dimensions. We conclude that embedding dimensions, both for event embedding and sequence embedding, contribute positively to the accuracy of the models.

U	Datasets					
	D9			D10		
	$CiT_s$	$CiT_{tm}$	$CiT_{ts}$	$CiT_s$	$CiT_{tm}$	$CiT_{ts}$
32	93.24	95.37	96.77	94.55	96.49	97.76
64	93.87	96.16	97.46	95.61	97.24	98.07
128	94.81	96.86	98.12	95.85	97.64	98.55
192	95.12	97.38	98.73	96.36	97.99	99.38
256	95.72	98.78	99.67	97.68	98.36	99.92

Table 8: AUC (%) vs. event sequences embedding dimensions ( $Em = 250$ )

## Other Deep Learning Algorithms

As the last part of the hyperparameter study, we train our approaches with different deep learning algorithms. We conduct experiments on three types of recurrent neural network units including LSTM, Gated Recurrent Unit (GRU), and simple Recurrent Neural Network (RNN). As shown in Table 9, LSTM outperforms both GRU and RNN, while GRU and RNN perform similarly. Therefore, in our implementation, all the *CiT* approaches are trained based on LSTM to achieve the best detection results.

Type	Datasets					
	D9			D10		
	$CiT_s$	$CiT_{tm}$	$CiT_{ts}$	$CiT_s$	$CiT_{tm}$	$CiT_{ts}$
LSTM	95.72	98.78	99.67	97.68	98.36	99.92
GRU	93.73	95.76	94.33	93.81	95.82	96.78
RNN	92.54	93.24	93.64	93.66	94.78	96.45

Table 9: AUC (%) vs. network hidden unit types ( $U = 256/Em = 250$ )

## 5.2.9 Efficiency and Scalability

In this chapter, we study the efficiency and scalability of *CiT* by first analyzing the training time for its three variations with five datasets (D1, D2, D7, D9, and D10), and the impact of hyperparameters with D9 and D10. Then we investigate the testing time for both translation and detection.

### Training Time

Figure 50 shows the training time of *CiT* on the five datasets. For fair comparison, we take the same number (50,000) of input pairs from each dataset. In general,  $CiT_{ts}$  (with both LSTM and Siamese) requires the longest training time. The color separation of the bars shows the training time for both translation and detection for this variation. However, in practice the translation and detection models can be trained in parallel to reduce the

overall training time. We can also observe that the corpus size and the length of sequences both affect the training time. Since the total number of input pairs is the same for different datasets, the corpus size becomes the main factor that impacts the overall training time. For instance, the corpus size in D7 is smaller and with shorter sequences, the training time is shorter than other datasets. Figure 51 studies how the size of training dataset (the number of input training pairs) affects the training time for the three variations. In general, training time increases when the size of the dataset increases. The longest time needed for training the largest dataset is 49 minutes per epoch in  $CiT_{ts}$ . Parallel training for both models at the same time will reduce the overall training time to 38 minutes per epoch.

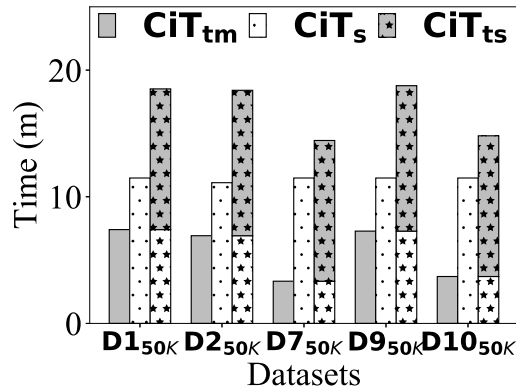


Figure 50:  $CiT$  efficiency study: Training Time. The results are obtained based on LSTM with  $E_m = 256$  and  $U = 250$ .

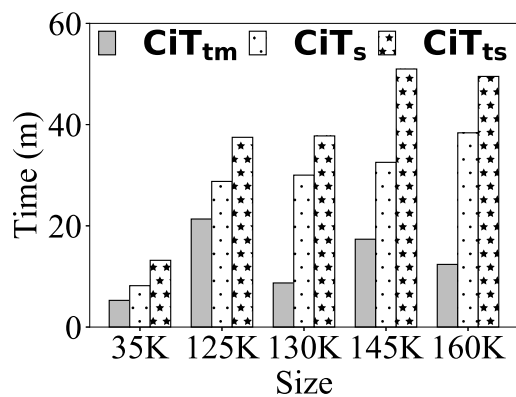


Figure 51:  $CiT$  efficiency study: Time vs. Training Pairs. The results are obtained based on LSTM with  $E_m = 256$  and  $U = 250$ .

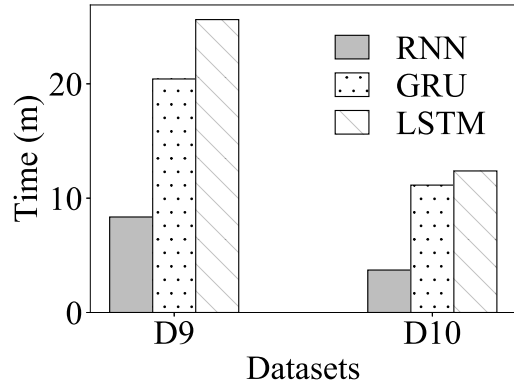


Figure 52: *CiT* efficiency study: Hidden Unit Type. The results are obtained based on LSTM with  $E_m = 256$  and  $U = 250$ .

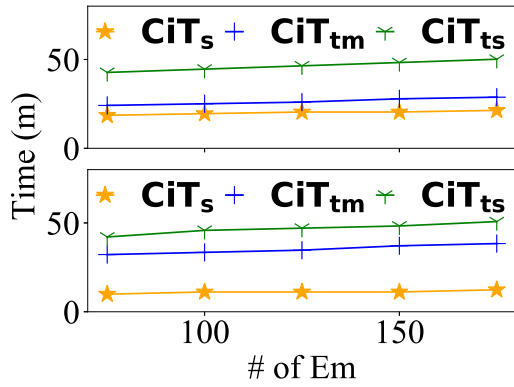


Figure 53: *CiT* efficiency study: Event Embedding Dimension; The top sub-figure shows the results from D9 dataset, and the bottom sub-figure shows the results from D10 dataset.

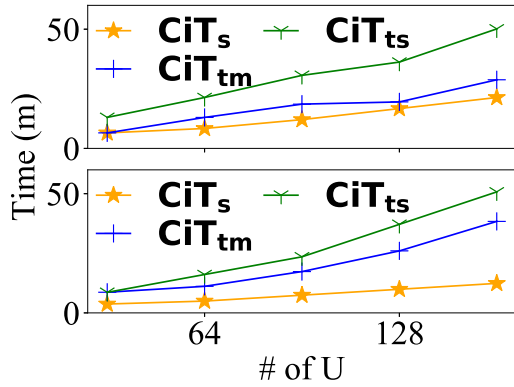
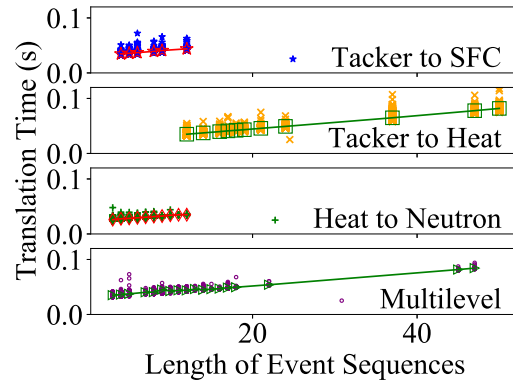


Figure 54: *CiT* efficiency study: Seq. Embedding Dimension; The top sub-figure shows the results from D9 dataset, and the bottom sub-figure shows the results from D10 dataset.

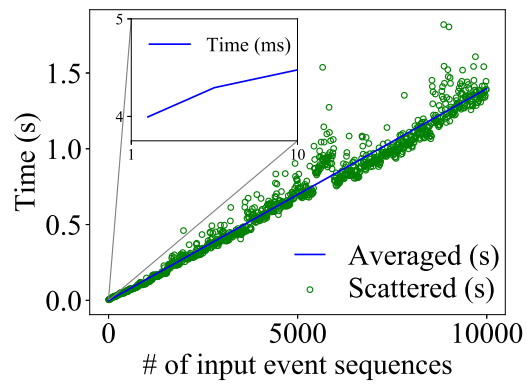
The training time for other hyperparameters, network hidden layer types, event embedding dimensions, and sequences embedding dimensions are shown in Figure 52, Figure 53, and Figure 54, respectively. We observe that LSTM requires longer training time than GRU and RNN due to its complexity (which also helps achieve better accuracy). The training time also increases linearly with both event ( $Em$ ) and sequence embedding ( $U$ ) dimensions, as presented in Figure 53 and Figure 54. This is expected as the number of embedding dimensions increases the increased complexity leading to a higher training time. As observed in Chapter 5.2.8, we can achieve a good model with 20 epochs of training. The overall training time is approximately three and half hours ( $= 11 * 20/60$ ) for obtaining a good  $CiT_s$  model with 50,000 training pairs. Since model training is a one-time effort and retraining of the models only happens after major changes or upgrades to the platform, we conclude our solution is scalable. Notably, we perform all the training on a server *without* GPUs and utilizing GPUs could significantly reduce the training time [6, 79].

### Testing Time

We investigate the testing time of  $CiT$  for both translation and inconsistency detection. Figure 55a shows the translation time required for each dataset. As a general trend, longer sequences naturally require longer translation time; however, our results show that even the longest sequence only takes 0.12s to finish translation. In Figure 55b, we evaluate the execution time of similarity score calculation versus the number of input event sequences by testing the trained  $CiT_s$  model with pairwise input sequences. In the zoomed in chart inside Figure 55b, we can observe the detection time is around  $4ms$  for a single event sequence pair input. Furthermore, the detection model only takes 1.5s to calculate similarity scores for 10,000 pairs of inputs. In contrast, according to our real-world data, a user-level operation would take OpenStack around 77 seconds on average to execute in a real cloud. Therefore, we conclude that  $CiT$  is an efficient solution for detecting inconsistency attacks.



(a) *CiT* translation time



(b) *CiT* detection time

Figure 55: The evaluation of testing time

# Chapter 6

## Discussion

### 6.1 Additional Use Cases

Besides inconsistency detection, the translation capability of *CiT* can be applied to other use cases. First, as *CiT* performs event-based detection, it can be leveraged for attack prevention at runtime, e.g., integrating with a *WSGI* middleware [104] (or similar middleware supports, e.g., Amazon Lambda Function [2], Google Cloud Metrics [29], and Azure Event Grid [52]) to intercept and deny events that would cause inconsistencies. Our efficiency results for detection (Chapter 5.2.9) demonstrate the feasibility of this use case. Second, using  $CiT_{tm}$  and  $CiT_{ts}$  (discussed in Chapter 4.6), we can translate VNF (e.g., Tacker)-level events to the lower levels, even before the actual events are triggered at those lower levels. This will enable us to proactively watch for (and deny) anomalous events without introducing extra delay. Third, *CiT* may help when data access is limited. As an NFV stack may involve multiple providers or domains, a user may not be allowed to access all the lower-level events and deployment details, which may be needed, e.g., for security verification, resource allocation verification, or network service compliance verification purposes [100]. In such cases,  $CiT_{tm}$  and  $CiT_{ts}$  can be utilized to translate the available events at a higher

level into the corresponding lower level events to facilitate those tasks (given that potential inconsistencies are not an issue). Alternatively, the lower-level providers may also translate their events into a higher level to share with the users.

## 6.2 Feasibility of Training

A well-known challenge faced by machine learning-based security solutions is to gather attack-free data for training. As shown in Chapter 5.2.7, the *CiT* models trained using data collected from our experimental testbed provided satisfactory results when tested on data from a completely different, real-world system. Such results confirm the feasibility of training *CiT* under a controlled environment and then applying it to a real-world NFV system (the training system could be made more similar to the real system than in our case, which will further improve the accuracy).

## 6.3 Employing Attention-based Translation Mechanisms

As shown in Chapter 5.2.1, *CiT* is not limited to any specific NMT implementation. It is possible to use other NMT mechanisms, such as RNN or attention [101], with *CiT* to detect inconsistencies. There exist several attention-based Seq2Seq translation mechanisms, such as the Transformer [101], Transformer-XL [12], BERT [13], and Reformer [39], which have proved to outperform RNN-based translation mechanisms in NMT. Our preliminary study with Transformer shows its potential to provide better accuracy than LSTM; however, it is significantly slower ( $\sim 10$  times) than LSTM, and hence we adopt LSTM in this work and will investigate Transformer in future work.



## 6.4 Adapting to other NFV Platforms

*CiT* can potentially be adapted to other NFV platforms (e.g., OSM [78] and OPNFV [77]) for inconsistency detection, since it has been designed based on the generic NFV architecture and deployment model. Most of the modules of *CiT* are platform agnostic except, as a one-time effort, the data processing module needs to be adjusted to extract platform specific event sequences through integration with either a generic middleware or logging system (e.g., syslog).

## 6.5 Limitations

The main limitations of our work are as follows. First, since *CiT* relies on the provider for collecting event sequences, how to ensure the integrity of such data (e.g., through trusted computing techniques) is a future direction. Second, *CiT* currently relies on the access to events at different levels, and anonymizing such events to avoid privacy concerns while still allowing the translation and detection is an interesting challenge. Finally, we have not considered potential adversarial attacks on the machine learning algorithms, and addressing this issue in the particular context of NFV is an interesting future direction.

# Chapter 7

## Related Work

This chapter reviews existing approaches to NFV modeling, NFV security, anomaly detection, and translation-based security solutions, and compares them to this thesis.

### 7.1 NFV Models

To the best of our knowledge, this is the first work proposing a concrete model for the deployment aspects of NFV. A few other models of NFV architecture are proposed for different use cases, e.g., mobile edge computing in 5G [22] and efficient VNF placement [54]. Pattaranantakul et al. [82] propose a framework to dynamically manage security functions in NFV. Hoang et al. [31] propose an extended NFV architecture that uses Tacker to support containers.

### 7.2 NFV Security

Unlike our work, which focuses on the inconsistencies, most existing studies on NFV security [26, 41, 83, 110] focus on issues related to virtualization. Lal et al. [41] propose to adapt several well-known best practices like VM separation, hypervisor introspection, and

remote attestation. Pattaranantakul et al. [83] adopt best practices like access control to address virtualization-related threats in NFV.

Most existing solutions in NFV that are related to inconsistency attacks (e.g., [25, 27, 50, 99, 102, 113, 114]) verify the NFVI-level configuration information (e.g., flow rules and flow classifier) while focusing on one particular level (mostly SFC). Particularly, ChainGuard [27] and SFC-Checker [99] both verify the correct forwarding behavior of SFCs. Moreover, vSFC [113] checks for a wide range of SFC violations (e.g., packet injection attacks, flow dropping, and path non-compliance). Additionally, there exist some works (e.g., [25, 50, 102, 114]) on verifying the functionality and performance of SFCs. For instance, vNFO [25] and SLAVerifier [114] verify a wide-range of SFC functionalities (e.g., performance and accounting). Guido et al. [50] propose an approach for generating optimal placement of SFCs based on given performance parameters (e.g., latency and CPU cycles) and reachability policies. Wang et al. [102] propose a framework to automatically detect the dependencies and conflicts between network functions. Unlike those approaches which rely on configurations, *CiT* is an event-based approach which means it can potentially catch an attack before it incurs any damage.

### **7.3 Anomaly Detection on Sequential Data**

There exist many works (e.g., [7, 16, 36, 85, 93, 105]) that conduct anomaly detection on sequential data (e.g., event logs, credit card transactions, and network traffic). Particularly, DeepLog [16] and Brown et al. [7] both leverage RNNs to detect anomalies in system logs. Also, Tiresias [93] utilizes RNNs to build a predictive model to detect future anomalies. Jurgovsky et al. [36] use the LSTM-based sequence classification to detect anomalies in credit card transactions. Radford et al. [85] propose a network traffic anomaly detection

solution by utilizing LSTM to learn and predict communications between two IPs. Similarly, Xiao et al. [105] utilize the semantic information of system call sequences for Android malware detection. Besides focusing on a different context (NFV), *CiT* applies the additional translation step before detection, which outperforms traditional anomaly detection (see Chapter 5.2.6).

## 7.4 Translation-based Security Approaches

Most of the existing security solutions that leverage neural machine translation (e.g., [15, 51, 105, 108, 115]) focus on binary code analysis, e.g., to support multiple hardware architectures like x86 and ARM. In particular, SAFE [51] leverages GRU RNN [10] and learns function embedding automatically where each assembly instruction is considered as a word and each sequence of instructions as a sentence. To find similar functions from binary code, INNEREYE [115] leverages LSTM and considers assembly instruction with its operands as a single word and the basic block as a sentence. Xu et al. [108] and Asm2vec [15] apply neural networks to translate binaries for code similarity detection. Although in a different context, those works show the potential of applying neural machine translation to security, and have inspired us for our work.

In summary, this is the first work proposing a concrete model for the deployment aspects of NFV. Furthermore, though inspired by those existing works, *CiT* differs from them because of its special focus on NFV, its event-based approach, and its use of neural machine translation for detection.

# Chapter 8

## Other Contributions

During this master thesis study, we also contributed to other projects as follows.

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

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

testbed, and experimentally evaluate its efficiency using both real and synthetic data.

The following lists the contributions to this research work towards the NFV data generation task.

- Developed a Java program for generating valid VNFFG descriptors which are used to deploy VNFFGs.
- Developed bash scripts for automatically deploying a number of VNFs and VNFFGs.
- Generated synthetic data for experiments using the NFV testbed.
- Assisted with writing the NFV data generation and its related challenges in the paper.

This research work is currently submitted to CloudCom'20.

## **8.2 NFV Testbed Deployment**

Deployed and actively maintained the NFV testbed (See Chapter 5.1) which is used by other NFV research projects. This chapter details the challenges encountered during the implementation of the NFV testbed.

### **8.2.1 Implementation Challenges**

- *Need for manual efforts* We utilized official OpenStack installation scripts [69] to implement the services required for the NFV testbed. The scripts stopped unexpectedly during the installation of services like Mistral and Barbican (dependencies for OpenStack Tacker) where inputs had to be provided during runtime. The services were manually installed.

- *Service version mismatch* A basic NFV implementation requires at least 14 OpenStack services operating together as a stack. But a version mismatch among services can cause the entire NFV deployment to collapse, and the troubleshooting becomes very challenging.
- *Conflicting service dependencies* The services involved in the NFV stack require dependencies which sometimes cause conflicts between each other during installation. We used Python virtual environments to keep the service dependencies independent from another.
- *Faulty quota management* During the large-scale deployment process where thousands of VNFs and VNFFGs need to be created, the tenants' resource quota allocation has to be adjusted. We were not able to update the resource quota of networking-sfc [76] component which is responsible for chaining VNFs. We were able to create only 10 VNFFGs per tenant. Hence, we increased the number of projects in order to be able to create more VNFFGs.
- *Undocumented deployment constraints* We developed Python and Java programs to automate the deployment of VNFs and VNFFGs by generating their corresponding descriptors. During the deployment, we faced constraints which are not officially documented thus resulting in deployment failures, such as (i) unable to chain VNFs with their management ports, (ii) unable to chain VNFs from different subnets, and (iii) the network traffic source must be within the same subnet as the VNFs. We validated all the generated VNFFG descriptors based on these constraints for a successful deployment.

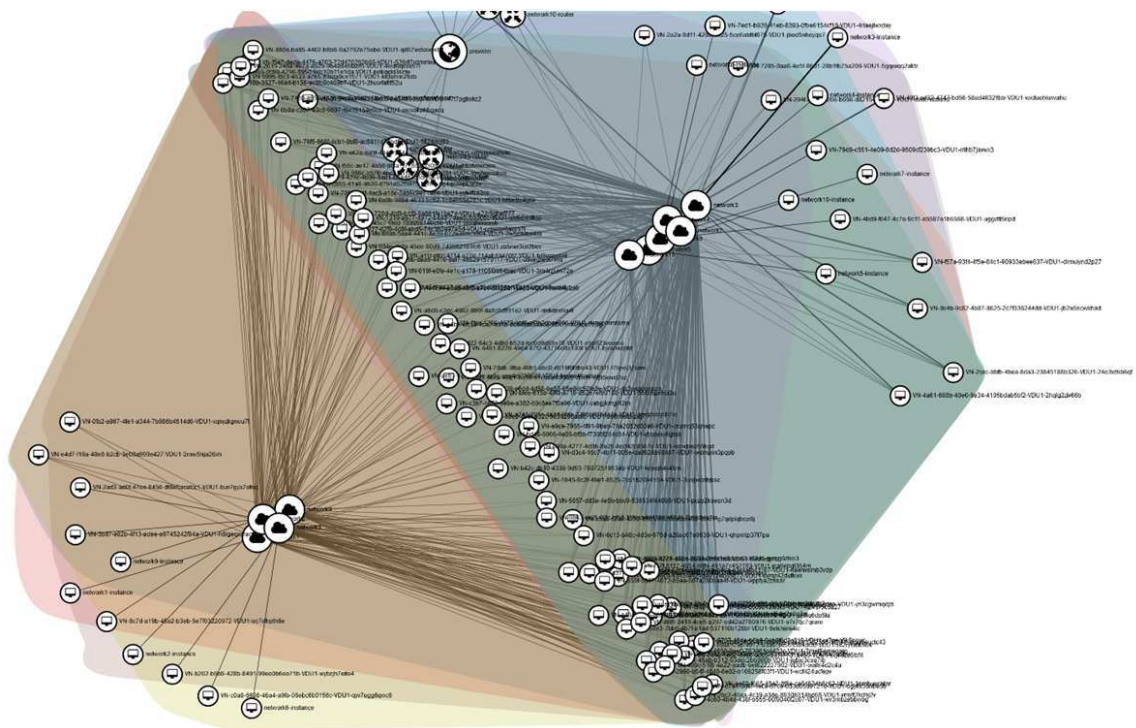


Figure 56: The topology view of VNFs implemented in our NFV tested from Horizon [66]



# Chapter 9

## Future Work and Conclusion

### 9.1 Future Work

As future work, in addition to formalizing the proposed NFV deployment model, we will extend our network-centric model to include other computing or storage managerial components. Furthermore, we intend to develop security verification tools based on open source NFV environments. We will also investigate the possibility of utilizing attention-based translation methodologies such as Transformer, BERT and Transformer-XL to further improve the performance and efficiency of CiT. Furthermore, we will also further enhance *CiT* to overcome its current limitations. For instance, since the results of the detection system entirely depends upon the quality of the datasets used, we would like to evaluate the data quality. Finally, we also intend to integrate CiT to other available open source NFV platforms.

## 9.2 Conclusion

In this thesis, we presented a multilevel NFV deployment model, which complements the ETSI architecture with essential details for exploring potential inconsistency vulnerabilities in NFV. Our model showed that the autonomous management components at different levels render cross-level inconsistencies an intrinsic threat to NFV. We validated our model by implementing an NFV testbed and concrete attack scenarios. Our study on the data collection paved the way for developing verification-based detection solution. Furthermore, we have proposed an event-based, Neural Machine Translation (NMT)-powered detection approach, namely, *CiT*, for cross-level inconsistency attacks in NFV. Specifically, we have leveraged the Long Short-term Memory (LSTM) model to translate the event sequences between different levels of an NFV stack. We have applied both similarity metric and Siamese neural network to compare the translated event sequences with the actual sequences to detect inconsistency attacks. As a proof of concept, we have integrated *CiT* into OpenStack/Tacker and conducted extensive experiments using both real and synthetic data to demonstrate the efficiency, accuracy, and robustness of our solution.

# Bibliography

- [1] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.
- [2] Amazon. AWS Lambda Features, 2020. Available at: <https://aws.amazon.com/lambda/>, last accessed on: July 24, 2020.
- [3] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- [4] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim. Proactive security analysis of changes in virtualized infrastructures. In *Proceedings of the 31st annual computer security applications conference*, pages 51–60, 2015.
- [5] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.
- [6] D. Britz, A. Goldie, M.-T. Luong, and Q. Le. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017.
- [7] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *Proceedings of the First Workshop on Machine Learning for Computing Systems*, pages 1–8, 2018.
- [8] R. Caruana, S. Lawrence, and C. L. Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*, pages 402–408, 2001.
- [9] K. Chen, Z. Zhang, J. Long, and H. Zhang. Turning from TF-IDF to TF-IGM for term weighting in text classification. *Expert Systems with Applications*, 66:245–260, 2016.
- [10] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

- [11] M. Creutz, T. Hirsimäki, M. Kurimo, A. Puurula, J. Pylkkönen, V. Siivola, M. Varjokallio, E. Arisoy, M. Saraçlar, and A. Stolcke. Morph-based speech recognition and modeling of out-of-vocabulary words across languages. *ACM Transactions on Speech and Language Processing (TSLP)*, 5(1):1–29, 2007.
- [12] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.
- [13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [14] R. Dey and F. M. Salemt. Gate-variants of gated recurrent unit (gru) neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pages 1597–1600. IEEE, 2017.
- [15] S. H. Ding, B. C. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.
- [16] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1285–1298, 2017.
- [17] ETSI. ETSI. <https://www.etsi.org/>.
- [18] ETSI. Network functions virtualisation - Use cases, 2013.
- [19] ETSI. Network functions virtualisation architectural framework, 2013.
- [20] ETSI. Network functions virtualisation - Report on SDN usage in NFV architectural framework, 2015.
- [21] ETSI. Network function virtualisation (NFV); Reliability; Report on the resilience of NFV-MANO critical capabilities, 2017.
- [22] ETSI. MEC in 5G networks, 2018.
- [23] ETSI. Network functions virtualisation (NFV) release 3; Management and orchestration; Architecture enhancement for security management specification, 2018.
- [24] I. Farris, T. Taleb, Y. Khettab, and J. Song. A survey on emerging sdn and nfv security mechanisms for iot systems. *IEEE Communications Surveys & Tutorials*, 21(1):812–837, 2018.

- [25] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar. Verifiable network function outsourcing: requirements, challenges, and roadmap. In *Proceedings of the workshop on Hot topics in middleboxes and network function virtualization*, pages 25–30, 2013.
- [26] M. D. Firoozjaei, J. P. Jeong, H. Ko, and H. Kim. Security challenges with network functions virtualization. *Future Generation Computer Systems*, 67:315–324, 2017.
- [27] M. Flittner, J. M. Scheuermann, and R. Bauer. ChainGuard: Controller-independent verification of service function chaining in cloud computing. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–7. IEEE, 2017.
- [28] G. Gardikis, K. Tzoulas, K. Tripolitis, A. Bartzas, S. Costicoglou, A. Lioy, B. Gaston, C. Fernandez, C. Davila, A. Litke, et al. SHIELD: A novel NFV-based cybersecurity framework. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6. IEEE, 2017.
- [29] Google. Google Cloud metrics, 2020.
- [30] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.
- [31] C.-P. Hoang, N.-T. Dinh, and Y. Kim. An extended virtual network functions manager architecture to support container. In *ICISS’18*, pages 173–176, 2018.
- [32] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [33] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *NDSS’15*, pages 8–11, 2015.
- [34] G. M. Insights. Network function virtualization (NFV) market size, 2020. Available at: <https://www.gminsights.com/industry-analysis/network-function-virtualization-nfv-market>, last accessed on: July 23, 2020.
- [35] Intel. Realising the benefits of network functions virtualisation in telecoms networks, 2014. Available at: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/benefits-network-functions-virtualization-telecoms-paper.pdf>, last accessed on: July 24, 2020.
- [36] J. Jurgovsky, M. Granitzer, K. Ziegler, S. Calabretto, P.-E. Portier, L. He-Guelton, and O. Caelen. Sequence classification for credit-card fraud detection. *Expert Systems with Applications*, 100:234–245, 2018.

- [37] Keras. Keras: The Python Deep Learning library. Available at: <https://keras.io/>, last accessed on: July 24, 2020.
- [38] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI'13*, pages 15–27, 2013.
- [39] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.
- [40] S. M. Lakew, M. Cettolo, and M. Federico. A comparison of transformer and recurrent neural networks on multilingual neural machine translation. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 641–652, Santa Fe, New Mexico, USA, Aug. 2018. Association for Computational Linguistics.
- [41] S. Lal, T. Taleb, and A. Dutta. NFV: Security threats and best practices. *IEEE Communications Magazine*, 55(8):211–217, 2017.
- [42] O. Levy and Y. Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014.
- [43] B. Li and L. Han. Distance weighted cosine similarity measure for text classification. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 611–618. Springer, 2013.
- [44] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI'15*, pages 499–512, 2015.
- [45] L. v. d. Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.
- [46] T. Madi, Y. Jarraya, A. Alimohammadifar, S. Majumdar, Y. Wang, M. Pourzandi, L. Wang, and M. Debbabi. ISOTOP: Auditing virtual networks isolation across cloud layers in openstack. *ACM TOPS*, 22(1):1:1–1:35, 2018.
- [47] 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 Transactions on Privacy and Security (TOPS)*, 22(1):1–35, 2018.
- [48] 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 *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 195–206, 2016.
- [49] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff. LSTM-based encoder-decoder for multi-sensor anomaly detection. *arXiv preprint arXiv:1607.00148*, 2016.

- [50] G. Marchetto, R. Sisto, J. Yusupov, and A. Ksentini. Virtual network embedding with formal reachability assurance. In *CNSM*, 2018.
- [51] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.
- [52] Microsoft. Azure Event Grid, 2020. Available at: <https://azure.microsoft.com/en-us/services/event-grid/>, last accessed on: July 24, 2020.
- [53] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [54] H. Moens and F. De Turck. VNF-P: A model for efficient placement of virtualized network functions. In *CNSM'14*, pages 418–423, 2014.
- [55] J. Mueller and A. Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *thirtieth AAAI conference on artificial intelligence*, 2016.
- [56] National Institute of Standards and Technology. CVE-2015-3456 Detail, 2015. Available at: <https://nvd.nist.gov/vuln/detail/CVE-2015-3456>, last accessed on: April 17, 2020.
- [57] National Institute of Standards and Technology. CVE-2015-7835 Detail, 2015. Available at: <https://nvd.nist.gov/vuln/detail/CVE-2015-7835>, last accessed on: April 17, 2020.
- [58] National Institute of Standards and Technology. CVE-2018-10853 Detail, 2018. Available at: <https://nvd.nist.gov/vuln/detail/CVE-2018-10853>, last accessed on: April 17, 2020.
- [59] NIST. The NIST definition of cloud computing, 2011.
- [60] Oasis. Topology and Orchestration Specification for Cloud Applications (TOSCA), 2013. Available at: <https://www.oasis-open.org/committees/tosca/>, last accessed on: July 30, 2020.
- [61] ONAP. Open Network Automation Platform. Available at: <https://www.onap.org/>.
- [62] OpenDaylight. OpenDaylight Project. Available at: <https://www.opendaylight.org/>.
- [63] OpenDaylight. CVE-2018-1078: OpenDaylight - Insecure behavior in node reconciliation process, 2018.

- [64] OpenStack. OpenStack. Available at: <https://www.openstack.org/>, last accessed on: July 24, 2020.
- [65] OpenStack. OpenStack Heat. Available at: <https://docs.openstack.org/heat/latest/>.
- [66] OpenStack. OpenStack Horizon. Available at: <https://docs.openstack.org/horizon/latest/>, last accessed on: July 24, 2020.
- [67] OpenStack. OpenStack Neutron. Available at: <https://docs.openstack.org/neutron/latest/>.
- [68] OpenStack. Openstack security advisories. Available at: <https://security.openstack.org/ossalist.html>.
- [69] OpenStack. OpenStack Training Labs. Available at: [https://docs.openstack.org/training\\_labs/](https://docs.openstack.org/training_labs/), last accessed on: July 24, 2020.
- [70] OpenStack. OpenStack Congress, 2015. Available at: <https://wiki.openstack.org/wiki/Congress>, last accessed on: July 24, 2020.
- [71] OpenStack. Stack update failed: port still in use, 2015. Available at: <https://bugs.launchpad.net/heat/+bug/1517355>, last accessed on: July 23, 2020.
- [72] OpenStack. Cinder fails to create volume with gateway time-out error under high load, 2016. Available at: <https://bugs.launchpad.net/mos/+bug/1653164>, last accessed on: July 23, 2020.
- [73] OpenStack. Heavy reading study on CSPs and OpenStack, 2016.
- [74] OpenStack. Rule:shared is not respected in port/subnet create, 2018. Available at: <https://bugs.launchpad.net/neutron/+bug/1808112>, last accessed on: July 23, 2020.
- [75] OpenStack. User is not allowed to create port with fixed IP on shared network via RBAC, 2019. Available at: <https://bugs.launchpad.net/neutron/+bug/1833455>, last accessed on: July 23, 2020.
- [76] OpenStack Networking-SFC. OpenStack Networking: Service Function Chaining. Available at: <https://docs.openstack.org/networking-sfc/latest/>, last accessed on: August 09, 2020.
- [77] OPNFV. OPNFV. Available at: <https://www.opnfv.org/>, last accessed on: July 24, 2020.
- [78] OSM. Open source MANO. Available at: <https://osm.etsi.org/>, last accessed on: July 24, 2020.



- [79] M. Ott, S. Edunov, D. Grangier, and M. Auli. Scaling neural machine translation. *arXiv preprint arXiv:1806.00187*, 2018.
- [80] Ovum. NFV deployments still on the rise, 2020. Available at: <https://www.ondia.com/resources/product-content/ovum-survey-results-on-nfv-adoption-spt002-000284>, last accessed on: April 12, 2020.
- [81] pandas. pandas - Python Data Analysis Library. Available at: <https://pandas.pydata.org/>, last accessed on: July 24, 2020.
- [82] M. Pattaranantakul, R. He, A. Meddahi, and Z. Zhang. SecMANO: Towards network functions virtualization NFV based security management and orchestration. In *IEEE Trustcom/BigDataSE/ISPA*, pages 598–605, 2016.
- [83] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, and A. Meddahi. NFV security survey: From use case driven threat analysis to State-of-the-art countermeasures. *IEEE Communications Surveys & Tutorials*, 20(4):3330–3368, 2018.
- [84] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.
- [85] B. J. Radford, L. M. Apolonio, A. J. Trias, and J. A. Simpson. Network traffic anomaly detection using recurrent neural networks. *arXiv preprint arXiv:1803.10769*, 2018.
- [86] J. Ramos et al. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. New Jersey, USA, 2003.
- [87] RedHat. The year of open source networking for CSPs, 2018. Available at: <https://www.redhat.com/en/blog/2018-year-open-source-networking-csps>.
- [88] R. Rehurek and P. Sojka. Gensim—statistical semantics in Python. *Retrieved from genism.org*, 2011.
- [89] N. Reimers and I. Gurevych. Optimal hyperparameters for deep LSTM-networks for sequence labeling tasks. *arXiv preprint arXiv:1707.06799*, 2017.
- [90] F. Reynaud, F.-X. Aguessy, O. Bettan, M. Bouet, and V. Conan. Attacks against network functions virtualization and software-defined networking: State-of-the-art. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 471–476. IEEE, 2016.
- [91] C. Sammut and G. I. Webb, editors. *Encyclopedia of Machine Learning and Data Mining*. Springer, 2017.

- [92] scikitlearn. scikit-learn: Machine Learning in Python. Available at: <https://scikit-learn.org/stable/>, last accessed on: July 24, 2020.
- [93] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini. Tiresias: Predicting security events through deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605, 2018.
- [94] M.-K. Shin, Y. Choi, H. H. Kwak, S. Pack, M. Kang, and J.-Y. Choi. Verification for NFV-enabled network services. In *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 810–815. IEEE, 2015.
- [95] S. W. Shin and G. Gu. Attacking software-defined networks: A first feasibility study. In *HotSDN'13*, pages 165–166, 2013.
- [96] L. T. Sudershan, M. Zhang, A. Oqaily, G. S. Chawla, L. Wang, M. Pourzandi, and M. Debbabi. Modeling NFV deployment to identify the cross-level inconsistency vulnerabilities. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 167–174. IEEE, 2019.
- [97] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.
- [98] Tacker. OpenStack Tacker. Available at: <https://wiki.openstack.org/wiki/Tacker>, last accessed on: July 24, 2020.
- [99] 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 *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 134–140. IEEE, 2016.
- [100] P. Twamley, M. Müller, P.-B. Bök, G. K. Xilouris, C. Sakkas, M. A. Kourtis, M. Peuster, S. Schneider, P. Stavrianos, and D. Kyriazis. 5GTANGO: An approach for testing nfv deployments. In *2018 European Conference on Networks and Communications (EuCNC)*, pages 1–218. IEEE, 2018.
- [101] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [102] Y. Wang, Z. Li, G. Xie, and K. Salamatian. Enabling automatic composition and verification of service function chain. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, pages 1–5. IEEE, 2017.
- [103] WordNet. WordNet: a lexical database for English. Available at: <https://wordnet.princeton.edu/>, last accessed on: July 24, 2020.

- [104] WSGI. Middleware and libraries for WSGI, 2016. Available at: <http://wsgi.readthedocs.io/en/latest/libraries.html>, last accessed on: February 15, 2020.
- [105] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah. Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications*, 78(4):3979–3999, 2019.
- [106] Z. Xing, J. Pei, and E. Keogh. A brief survey on sequence classification. *ACM Sigkdd Explorations Newsletter*, 12(1):40–48, 2010.
- [107] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu. Attacking the brain: Races in the SDN control plane. In *USENIX Security’17*, pages 451–468, 2017.
- [108] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 363–376, 2017.
- [109] Y. Xu, Y. Liu, R. Singh, and S. Tao. Identifying SDN state inconsistency in OpenStack. In *SOSR’15*, page 11, 2015.
- [110] W. Yang and C. Fung. A survey on security in network functions virtualization. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 15–19. IEEE, 2016.
- [111] W.-t. Yih, K. Toutanova, J. C. Platt, and C. Meek. Learning discriminative projections for text similarity measures. In *Proceedings of the fifteenth conference on computational natural language learning*, pages 247–256, 2011.
- [112] F. Z. Yousaf, M. Bredel, S. Schaller, and F. Schneider. NFV and SDN—key technology enablers for 5G networks. *IEEE Journal on Selected Areas in Communications*, 35(11):2468–2478, 2017.
- [113] X. Zhang, Q. Li, J. Wu, and J. Yang. Generic and agile service function chain verification on cloud. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2017.
- [114] 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.
- [115] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.