Security Auditing for Network Function Virtualization (NFV) and Microservices

Alaa Oqaily

A THESIS

IN THE DEPARTMENT OF

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy (Information and Systems)

AT CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

August 2025

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Alaa Oqaily**

Entitled: **Security Auditing for Network Function Virtualization (NFV) and Microservices**

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

Doctor of Philosophy (Information and Systems)

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. Rongxing Lu

______________________Examiner
Dr. Otmane Ait Mohamed

______________________ Examiner
Dr. Suryadipta Majumdar

______________________ Examiner
Dr. Jun Yan

______________________ Thesis Supervisor
Dr. Lingyu Wang and Dr. Yosr Jarraya

Approved by ______________________
Dr. Farnoosh Naderkhani, Graduate Program Director

28/05/2025 ______________________
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Security Auditing for Network Function Virtualization (NFV) and Microservices

Alaa Oqaily, Ph.D.

Concordia University, 2025

Advancements in virtualization technologies and frameworks have profoundly transformed the deployment and management of networks and applications. Network Functions Virtualization (NFV), for instance, has revolutionized the networking landscape by decoupling Network Functions (NFs) from dedicated hardware, offering enhanced flexibility, scalability, and cost-efficiency. In parallel, the microservice architecture has transformed cloud application development by structuring it as a collection of small, loosely coupled services. This design enables independent development, deployment, and scaling of individual functionalities, promoting agility and resilience in modern cloud environments. However, despite their benefits, NFV and microservices introduce novel security and privacy challenges. For instance, attackers could exploit inconsistencies across different system layers to bypass security mechanisms, resulting in cloud-level breaches that remain undetected by NFV tenants. Similarly, the distributed nature of microservice architectures expands the attack surface and complicates the management of data privacy across multiple independent services. To facilitate their adoption, robust security auditing solutions are crucial for ensuring compliance and detecting potential breaches. However, existing security auditing solutions face significant challenges. They often fall short in verifying NFV security because they focus on individual levels, which can lead to overlooking cross-level inconsistencies or vulnerabilities. As a result, potential breaches may go undetected, since issues at one level might not be visible or addressed by audits focused solely on other levels. Moreover, verifying each level separately would be both expensive and impractical. Additionally, the complexity and scale of these virtual environments can render verification solutions, such as formal security checks, prohibitively expensive. This could lead to delays in detecting misconfigurations, creating a significant window of vulnerability where services or infrastructure remain exposed to potential attacks. Moreover, the distributed nature of microservices, combined with privacy concerns, makes it difficult to centralize data for security verification using existing solutions. This thesis presents novel solutions for security verification in virtualized environments, addressing the aforementioned challenges. Firstly, it introduces NFVGuard+, a cross-level security verification approach that efficiently ensures security throughout the NFV stack by conducting resource-intensive verification at one level and then propagating the results to other levels using relatively lightweight consistency checks. Furthermore, its practicality is ensured by automating key verification processes by leveraging a novel Entity-Relationship (ER) model of the NFV stack. Secondly, it presents MLFM, an approach that combines the efficiency of Machine Learning (ML) with the rigor of Formal Methods (FM) to enable fast and provable detection of security violations in large NFV environments. The core idea is an iterative teacher-learner interaction, where FM (the teacher) progressively refines verification results to generate representative training data, while ML (the learner) utilizes this data to build increasingly accurate models. This interaction allows a relatively small subset of configuration data to train an effective ML model, which can then be used to prioritize verification efforts on configurations most likely to contain security violations. Finally, it introduces FLFM, a Federated Learning (FL)-guided Formal Method (FM) approach for the security verification of microservice-based cloud applications. FLFM enables scalable and decentralized verification while preserving privacy by eliminating the need for applications to share their sensitive local data.

Acknowledgments

The successful completion of this thesis reflects a collaborative journey shaped by the insights, efforts, and unwavering support of many individuals. I am truly grateful to everyone who contributed to this endeavor and stood by me throughout the process.

First and foremost, I offer my deepest thanks and praise to Allah, whose infinite mercy and blessings have guided and sustained me throughout this journey. By His will and grace, I have reached this stage, and I am truly grateful for the strength and clarity He granted me in times of hardship. After the blessings of His Majesty Allah, I would like to extend my heartfelt gratitude to my supervisor, Dr. Lingyu Wang. His continuous support, insightful guidance, and unwavering encouragement have been a cornerstone of my academic journey. His mentorship has not only deepened my understanding of the field but also inspired me to persevere through the many challenges of doctoral research. I am truly grateful for the opportunity to work under his supervision.

I also wish to sincerely thank my co-supervisor, Dr. Yosr Jarraya, for her continuous support and guidance throughout my Ph.D. journey. Her expertise, thoughtful feedback, and unwavering dedication played a crucial role in the progress and success of my research, and I am truly grateful for her meaningful contributions and commitment.

I am also deeply grateful to my colleagues at the Audit Ready Cloud research group for an inspiring seven-year journey marked by collaboration, innovation, and shared discovery in cutting-edge research areas. Being part of such a committed and talented team has played a pivotal role in shaping my academic and professional development. Specifically,

I am immensely thankful to Sudershan Lakshmanan, Dr. Mohammad Ekramul Kabir, Dr. Mengyuan Zhang, and Dr. Makan Pourzandi for the unwavering support, encouragement, understanding, and collective commitment to advancing knowledge, which have been instrumental in driving our collaborative success and enriching this journey.

Finally, I would like to express my heartfelt and profound gratitude to my family for their unwavering support, love, and encouragement throughout my entire Ph.D. journey. Their constant presence, understanding, and belief in me have been a source of strength and motivation, especially during the most challenging moments. Whether through offering a listening ear, providing emotional comfort, or simply being there when I needed them, my family has been my anchor. Without their support, this achievement would not have been possible, and I am forever grateful for everything they have done for me.

Contents

List of Figures **xi**

List of Tables **xvi**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Research Contributions 2

1.3.1 Cross-Level Security Verification System for Network Functions Virtualization 3

1.3.2 Machine Learning Meets Formal Method for Faster Identification of Security Breaches in Network Functions Virtualization 4

1.3.3 Security Verification for Microservices Using Federated Learning-Guided Formal Method 5

1.4 Relationships between the Research Topics 7

1.5 Thesis Structure 8

2 Related Work **10**

2.1 Security Verification System for Network Functions Virtualization 10

2.2 Security Verification for NFV Using Machine Learning and Formal Method 12

2.3 Security Verification for Microservices Using Federated Learning-Guided Formal Method . . . 13

3 Cross-Level Security Verification System for Network Functions Virtualization 17

3.1 Introduction . . . 17

3.2 Preliminaries . . . 21

3.2.1 Background on NFV . . . 21

3.2.2 Security Properties for NFV . . . 21

3.2.3 Challenges to Cross-Level Security Verification . . . 23

3.2.4 Threat Model . . . 25

3.3 Overview . . . 26

3.4 ER Model Construction and Consistency Property Identification . . . 27

3.4.1 Constructing the Entity Relationship (ER) Model . . . 27

3.4.2 Automated Consistency Property Derivation . . . 29

3.5 Cross-Level Security Verification . . . 32

3.6 Application to OpenStack/Tacker . . . 38

3.6.1 Deploying the NFV Testbed . . . 38

3.6.2 NFVGuard+ Implementation . . . 42

3.7 Experiments . . . 43

3.7.1 Experiments with Synthetic Data . . . 43

3.7.2 Experiments with Real Data . . . 54

3.8 Discussion . . . 55

3.9 Summary . . . 59

4 Machine Learning Meets Formal Method for Faster Identification of Security Breaches in Network Functions Virtualization 61

4.1 Introduction 61

4.2 Preliminaries 64

4.3 Methodology 67

4.3.1 Overview 67

4.3.2 Iterative Teacher (FM)-Learner (ML) Interaction 69

4.3.3 MLFM Algorithm and Use Cases 72

4.4 Implementation 74

4.5 EXPERIMENTS 77

4.5.1 Datasets and Experimental Settings 77

4.5.2 Experimental Results 78

4.6 Discussion 83

4.7 Summary 85

5 Security Verification for Microservices Using *F*ederated *L*earning-Guided *F*ormal *M*ethod 87

5.1 Introduction 87

5.2 Preliminaries 90

5.2.1 XGBoost 90

5.2.2 Federated Learning 91

5.2.3 Threat Model and Assumptions 94

5.3 Horizontal FLFM (H-FLFM) Methodology 95

5.3.1 Overview 95

5.3.2 Training Stage - Local Model Training 96

5.3.3 Training Stage - Global Model Learning 98

5.3.4 Application Stage 100

5.4 Vertical FLFM (V-FLFM) Methodology 100

5.4.1 Overview 100

5.4.2 Training Stage - Training Data Building 101

5.4.3 Training Stage - Global Model Learning 103

5.4.4 Application Stage 104

5.5 Implementation 104

5.6 Experiment 107

5.6.1 Investigated Properties 107

5.6.2 Experimental Settings and Datasets 110

5.6.3 H-FLFM Experimental Results 111

5.6.4 V-FLFM Experimental Results 116

5.7 Summary 120

6 Conclusion 121

Bibliography 122

List of Figures

Figure 3.1 A motivating example illustrating the challenges of cross-level security verification in NFV and our ideas. 18

Figure 3.2 The multilevel NFV model [1]. 22

Figure 3.3 An overview of the NFVGuard+ approach. 25

Figure 3.4 The ER model of the NFV stack. 30

Figure 3.5 Thumbnail of the ER model showing entities for verifying VNFFG configuration consistency property at L1. 36

Figure 3.6 The process of generating VNF and VNFFG/NS TOSCA template descriptors. 40

Figure 3.7 The topology of our NFV testbed (left) consisting of 20 tenants, 200 VNFFGs, and 200 VNFs and detailed view (in Horizon [2]) of an attack scenario similar to the motivating example in Section 3.1 (right). 40

Figure 3.8 Verification performance for the consistency properties while varying the number of service chains. 46

Figure 3.9 Verification time for the topology consistency properties in case of compliance (left), in case of reporting the first breach verifying between levels L2/L3 (middle), and in case of reporting the first breach verifying between L1/L2 (right). 47

Figure 3.10 Verification time for the security properties virtual resource isolation (left) and mapping unicity VLANs-VXLANs (middle) in case of compliance and in case of reporting the first breach. Verification time for finding all compliance breaches (10 breaches) for the consistency property L1/L2 using SAT and ALLSAT solvers (right). 48

Figure 3.11 Verification time for the topology consistency properties, virtual resource isolation, and mapping unicity VLANs-VXLANs in case of reporting all compliance breaches using ALLSAT solver, with (left) reporting all breaches for verifying between levels L2/L3, (middle) reporting all breaches for verifying between L1/L2, and (right) reporting all breaches for verifying virtual resource isolation and mapping unicity VLANs-VXLANs. 49

Figure 3.12 Verification time for reporting all breaches for the security property mapping unicity VLANs-VXLANs while varying the number of breaches (left) and the time for parallelizing the verification of the virtual resource isolation property (right). 51

Figure 3.13 Comparing the verification time of the multi-level security property without (the grayscale bar) and with (the bar with patterns) the utilization of ER model. 52

Figure 3.14 Verification performance for the consistency property L1/L2 using ALLSAT and SAT solvers. 53

Figure 3.15 The topology of a part of a real cloud data center operating NFV used in our experiments. 55

Figure 4.1 Motivating example 63

Figure 4.2 ETSI NFV reference architecture [3] (left) and an example NFV deployment corresponding to the motivating example (right) 65

Figure 4.3 Two example NFV security properties: *Mapping unicity VLANs-VXLANs* (left) and *No VNFs co-residence* (right) (shaded nodes indicate violations) 66

Figure 4.4 Overview of the MLFM approach 68

Figure 4.5 An example of the iterative teacher (FM)-learner (ML) interaction . 70

Figure 4.6 The MLFM system architecture 74

Figure 4.7 Recall and F1 score for combinations of ML models and sampling methods, trained on 20% of dataset DS1 for property P1 (a and b) and on DS2 for P2 (c and d) 79

Figure 4.8 Running time of MLFM for combinations of ML models and sampling methods, with 20% of training data under P1 (a) (b), or P2 (c) (d), for both use cases 80

Figure 4.9 Running time of MLFM vs. the baseline (FM only) under property P1 (a) and (b) or P2 (c) and (d), using different percentages of training data either by changing the sample size m (a) and (c) or by changing the number of iterations n (b) and (d) 81

Figure 4.10 The time (in minutes) for identifying different percentages of violations by MLFM and the baseline for P1 (a) or P2 (b). The tradeoff between running time and recall values of MLFM and the baseline for partial verification of P1 (c) or P2 (d) 83

Figure 4.11 The time (in minutes) for identifying different percentages of violations by NOD [4] and by MLFM integrated with NOD, using 25,000 (a) and 50,000 (b) records 83

Figure 5.1 Motivating example 89

Figure 5.2 An example illustrating XGBoost training 92

Figure 5.3 An example illustrating XGBoost training in the HFL (left) and VFL (right) settings 93

Figure 5.4 Overview of the horizontal FLFM approach 97

Figure 5.5 An example of the horizontal FLFM training 98

Figure 5.6 Overview of the vertical FLFM approach 102

Figure 5.7 An example of the vertical FLFM training 103

Figure 5.8 The FLFM architecture 105

Figure 5.9 The execution time (in hours) (a) and recall (b) for identifying violations in datasets with different levels of heterogeneity by H-FLFM and by MLFM [5] 113

Figure 5.10 The execution time of H-FLFM (in hours) for different sample sizes and for various numbers of iterations (aggregation and local iterations) (a). Execution time (in minutes) of H-FLFM using the best performing parameters (i.e., m = 250 and n = 10), while varying the number of aggregation iterations and compared to the execution time of MLFM using the same applicable parameters (b) 115

Figure 5.11 The execution time (in minutes) for identifying different percentages of violations by H-FLFM and MLFM (a). The time (in minutes) for identifying different percentages of violations by H-FLFM under different scenarios (b) 116

Figure 5.12 The percentage of records shared with the central authority by V-FLFM and the baseline approach for the partial verification case (a) and the recall of V-FLFM and the baseline approach (b) 117

Figure 5.13 The percentage of records shared with the central authority by V-FLFM and the baseline approach for the priority-based verification case (a) and the execution time (in minutes) for identifying different percentages of violations by V-FLFM and the baseline approach (b) 118

Figure 5.14 Percentage of shared records by V-FLFM and the baseline approach for partial verification (a) and priority-based verification (b) using different datasets, and using different datasets, and the recall of V-FLFM and the baseline approach using different datasets (c) 119

List of Tables

Table 1.1 List of acronyms used in thesis and their terminology 9

Table 2.1 Comparing our solution with existing solutions. 12

Table 2.2 Comparing our solution with existing solutions. 16

Table 3.1 Examples of NFV security properties [6] 24

Table 3.2 Example of consistency properties identified from the ER model entities and relationships. 31

Table 3.3 An excerpt of the data sources for some of the entities in the ER model, along with a description of the types of data they contain. 33

Table 3.4 Example property instances for evaluating the effectiveness of NFV-Guard+. 44

Table 3.5 The experimental results of NFVGuard+ for the real data. The average time, CPU, and memory required for the verification of three sample NFV security properties, i.e., VNFs co-residence, virtual resource isolation, and mapping unicity VLANs-VXLANs, based on real data. 55

Table 5.1 Main parameters evaluated in the experiments. 111

Chapter 1

Introduction

1.1 Motivation

The rise of NFV and microservices represents a fundamental shift toward modular, software-driven architectures, replacing rigid, hardware-bound systems with agile and dynamic solutions that enable greater flexibility and scalability. However, their significant advantages come at the cost of increased complexity, which introduces novel security and privacy challenges. For instance, in NFV systems, security threats can span across multiple levels, each managed by autonomous managerial components, making traditional, level-specific verification methods insufficient. Similarly, the distributed nature of microservices expands the attack surface and increases the risk of misconfigurations compared to monolithic architectures, while also raising privacy concerns that hinder centralized security verification. Existing approaches often fall short in delivering scalable, efficient, and cross-level verification without imposing high computational costs or compromising data privacy. These challenges highlight the pressing need for novel security verification techniques that can address the complexity and scale of modern virtualized systems while maintaining accuracy and preserving privacy. This research aims to explore these challenges and propose effective strategies to ensure the security and reliability of virtualized services.

1.2 Problem Statement

In this thesis, we propose security auditing solutions to ensure the security of virtualized network functions and microservices architecture. Specifically, we propose a formal cross-level security verification solution to overcome the limitations of traditional level-specific approaches. To enhance efficiency, our method utilizes consistency check results to perform verification at a single level and propagates these results across other levels, significantly reducing overhead while preserving verification accuracy. Moreover, to address scalability challenges, we propose a Machine Learning–Formal Method (MLFM) approach that combines the efficiency of Machine Learning (ML) with the rigor of Formal Methods (FM) for fast and provable detection of security violations. In particular, FM serves as the verifier, guiding the generation of high quality training data to build an effective ML model. This model is then used to identify configurations likely to violate security properties and prioritize them for formal verification, significantly improving the speed of detecting violations without compromising accuracy. Finally, to address the scalability and privacy concerns in microservice-based architectures, we present FLFM, a Federated Learning–guided Formal Method approach. FLFM allows decentralized security verification by enabling applications to collaboratively train models without sharing sensitive local data, ensuring privacy while still prioritizing high-risk configurations for formal analysis. Together, these contributions offer a scalable, efficient, and privacy-aware solution for security verification in both NFV and microservice-based cloud environments.

1.3 Research Contributions

In the following, we outline the key research contributions presented in this thesis. First, we propose NFVGuard+, a cross-level security verification system for Network Functions Virtualization (NFV). Second, we introduce MLFM, a Machine Learning–Formal Method approach for faster identification of security breaches in NFV. Finally, we present FLFM, a Federated Learning–guided Formal Method solution designed for security verification in microservice-based environments

1.3.1 Cross-Level Security Verification System for Network Functions Virtualization

Network Functions Virtualization (NFV) is a popular solution for providing multi-tenant network services on top of existing cloud infrastructures in an agile and cost-effective manner. However, as NFV employs multiple levels of virtualization, it also introduces novel security challenges, such as cloud-level security breaches that are invisible to NFV-level tenants. Towards verifying the security of NFV across all the levels (a.k.a. cross-level security verification), existing solutions are mostly insufficient, as each such solution typically only focuses on one specific level (e.g., cloud, SDN, or SFC), and verifying every level separately would be expensive or even infeasible. In this work, we propose an efficient and practical system, *NFVGuard+*, for cross-level security verification for NFV. Particularly, the efficiency of *NFVGuard+* is achieved by first performing the costly security verification at one level, and then extrapolating the verification result to other levels through conducting relatively lightweight consistency checks. Chapter 3 details our work on cross-level security verification for NFV.

Specifically, the main contributions of this work are as follows:

1. To the best of our knowledge, we are the first to propose a cross-level security verification system for NFV that automates the identification of properties and their data sources, thereby reducing the need for human intervention.

2. We are also the first to capture knowledge about system entities and their relationships across different levels of the NFV stack by developing an Entity Relationship

(ER) model for NFV. Additionally, we offer a concrete guideline for effectively identifying security properties using the ER model, which could be valuable for developing other security measures for NFV beyond just security verification.

3. We implement our solution and integrate it into a real NFV testbed built on OpenStack/Tacker [2], a widely used platform for deploying NFV [68]. Furthermore, we conduct experimental evaluations using both synthetic and real data, showcasing the efficiency and practicality of our solution.

1.3.2 Machine Learning Meets Formal Method for Faster Identification of Security Breaches in Network Functions Virtualization

By virtualizing proprietary physical devices, Network Functions Virtualization (NFV) enables agile and cost-effective deployment of network services on top of an existing cloud infrastructure. However, the added complexity also increases the chance of misconfigurations that could leave the services or infrastructure vulnerable to security threats. To that end, formal method-based security verification is a standard solution for providing rigorous mathematical proofs that the configurations satisfy the desired security properties, or the counterexamples (i.e., misconfigurations). Nonetheless, a major challenge is that the sheer scale of large NFV environments can render formal security verification so costly that the significant delays before misconfigurations can be identified may leave a wide attack window. In this work, we propose a novel approach, *MLFM*, that combines the efficiency of Machine Learning (ML) and the rigor of Formal Methods (FM) for fast and provable identification of misconfigurations violating security properties in NFV. Our key idea lies in an iterative teacher-learner interaction in which the teacher (FM) can gradually (over several iterations) provide more representative verification results as training data, while the learner (ML) can leverage such data to gradually obtain more accurate ML models. As a result, a small portion of the configuration data will be enough to obtain a relatively accurate

ML model, which can then be applied to the remaining data to prioritize the verification of data that are more likely to cause violations. Chapter 4 further details our approach of combining ML and FM for fast and provable identification of security breaches in NFV.

In summary, the main contributions of this work are:

1. To the best of our knowledge, we are the first to integrate Machine Learning (ML) with Formal Methods (FM), namely MLFM, combining the rigor of FM, which is crucial for proving security compliance, with the efficiency of ML, which is essential for handling large NFV environments, to prioritize verification tasks in NFV.

2. To implement MLFM, we design an iterative teacher-learner interaction approach, supported by a detailed algorithm. The methodology is realized through a constraint satisfaction problem solver, Sugar [67], along with several well-known machine learning algorithms (decision tree, random forest, support vector machine, and XGBoost) and sampling techniques (uncertainty sampling and query-by-committee) borrowed from the active learning literature [46] for selecting representative data records.

3. We conduct experimental evaluations of our work across two distinct use cases: one focused on minimizing verification time and the other on ensuring result completeness. The experimental results showcase the advantages of our work by detecting violations much more faster than the baseline FM [67] and further enhancing the efficiency of a state-of-the-art security verification tool [4].

1.3.3 Security Verification for Microservices Using Federated Learning-Guided Formal Method

The microservice architecture divides a single application into loosely coupled, autonomous microservices to allow for independent development, deployment, and scaling of different functionalities. The architecture is widely adopted in modern cloud environments due to its numerous benefits, such as enhanced scalability, flexibility, and cost efficiency in application development and maintenance. On the other hand, the sheer scale and distributed nature of microservice-based applications may also lead to novel challenges for existing security solutions. Particularly, the standard practice of using formal methods to provide rigorous mathematical proof about security compliance may face two major challenges in the context of microservices. First, large scale microservice applications can cause formal methods to become very slow in identifying security breaches, which may leave a wide attack window. Second, the prohibitive overhead and potential privacy concerns may both prevent the collection of data from all the microservices for performing security verification at a central location. In this work, we propose *FLFM*, a novel approach that combines the efficiency and privacy-friendliness of Federated Learning (FL) and the rigor of formal method (FM) for security verification in microservices. Specifically, FLFM works in two stages. First, each application samples a small but representative subset of its configuration data, and then labels such data using FM verification. This allows a relatively accurate FL model to be jointly trained by all the applications using only a small subset of data from each application. Second, the FL model can then be applied by each application to its remaining data in order to "guide" the FM verification for identifying security breaches faster, through prioritizing more suspicious candidates. Chapter 5 further describes our idea of utilizing the efficiency and privacy-friendliness of FL and the rigor of formal method (FM) for security verification in microservice.

Particularly, the main contributions of this work are as follows:

1. We propose a novel approach that leverages Federated Learning (FL) to guide formal methods (FM), namely FLFM, for faster identification of security breaches in microservice applications, while preserving the privacy of local data.

2. We provide comprehensive methodologies for both horizontal and vertical Federated

Learning (FL) scenarios and implement our solution using a federated XGBoost algorithm [109], an uncertainty sampling technique [113], and the Sugar constraint satisfaction problem solver [67].

3. We conduct experimental evaluations of FLFM and compare its performance with the centralized MLFM approach, demonstrating its effectiveness.

1.4 Relationships between the Research Topics

In the following, we outline the relationships between the three research topics and how they were identified. The primary objective of the first research topic was to develop an efficient and practical cross-level security auditing solution for verifying multiple levels of the NFV stack using formal methods. However, a key limitation of this approach is scalability. The complexity of formal security verification methods may restrict their ability to handle the sheer scale of virtualized services. This, in turn, could lead to significant delays in detecting misconfigurations and security threats, creating an extended attack window. Additionally, the inherent complexity of formal methods leaves minimal room for further performance optimizations. This leads to the second research topic, which leverages the efficiency of machine learning to create a scalable and provable solution for the faster identification of security breaches in NFV, while preserving the rigor of formal methods, which is crucial for ensuring security compliance. Finally, while this solution is effective in centralized environments like NFV, it is inadequate for distributed applications such as microservices. This leads to the third research topic, which leverages the efficiency and privacy benefits of Federated Learning (FL) alongside the rigor of formal methods (FM) to address the challenges posed by distributed environments. In summary, the three topics of this thesis are interrelated and serve as complementary elements of a unified solution, aimed at achieving secure, efficient, and privacy-preserving security verification for virtual environments.

1.5 Thesis Structure

This thesis is structured into six chapters. Chapter 1 provides an introduction to the research. Chapter 2 reviews the relevant literature. Chapter 3 presents the results of our cross-level security verification system for network functions virtualization. Chapter 4 details our approach to integrating formal methods with machine learning to prioritize verification tasks in NFV. Chapter 5 focuses on leveraging federated learning to enhance FM for faster detection of security breaches in microservice applications. Finally, Chapter 6 concludes the thesis, summarizing key findings and contributions. Additionally, Table 1.1 lists and defines the terminologies used throughout this thesis in alphabetical order.

Acronym	**Terminology**
CP	Connection Point
DPI	Deep Packets Inspector
ETSI	European Telecommunications Standards Institute
FC	Flow Classifier
FL	Federated Learning
FM	Formal Method
FW	Firewall
GDPR	General Data Protection Regulation
IDS	Intrusion Detection System
MANO	Management and Orchestration
ML	Machine Learning
MS	Microservice
NFP	Network Forwarding Path
NFV	Network Function Virtualization
NFVO	NFV Orchestrator
NLP	Natural Language Processing
NS	Network Service
NSD	Network Service Descriptor
PPG	PortPair Group
SDN	Software Defined Networking
SFC	Service Function Chain
VDU	Virtual Deployment Unit
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFD	VNF Descriptor
VNFFGD	VNFFG Descriptor
VNFFG	Virtual Network Function Forwarding Graph
VNFM	VNF Manager

Table 1.1: List of acronyms used in thesis and their terminology

Chapter 2

Related Work

In this chapter, we review related works of prior research on our identified problem areas.

2.1 Security Verification System for Network Functions Virtualization

Most existing security verification solutions (e.g., [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]) in NFV focus on the verification of one particular level (mostly SFC). In particular, ChainGuard [12], SFC-Checker [14], Cohen et al. [19], and AuditBox [20], all verify the correct forwarding behavior of SFCs. Other solutions, including NFVSense [9], CloudVaults [11], APPD [22], and Cheng et al. [10], focus on SFC integrity verification. vSFC [13] verifies various SFC violations (e.g., packet injection attacks and path non-compliance) and vHSFC [8] utilizes a lightweight Verified Routing Protocol (VRP) to detect various hybrid SFC violations and attacks. EnsureS [7] introduces an SFC path validation model that employs batch hashing and tag verification. VeriNeS [21] proposes a runtime verification framework for detecting anomalies in network services. In contrast, Zoure et al. [23] investigate NFV network service anomalies and the challenges in achieving verification.

Several solutions (e.g., [16, 17, 15, 18]) focus on verifying SFCs functionality and performance. They cover a wide range of verification aspects, such as performance and accounting [16], SLA-related performance properties [17], verification of reachability policies [18], and detection of dependencies and conflicts between network functions [15]. Unlike all those works, the main focus of our approach is to ensure the security of an NFV stack at all levels. Also, unlike us, most of those works do not formally model the verification problem.

There are a few solutions (e.g., [24, 25]) that tackle the multi-level aspect of NFV. Lakshmanan et al. [25] propose employing Neural Machine Translation (NMT) to detect cross-level inconsistency attacks. However, their utilization of NMT for detection is considered less reliable in terms of accuracy compared to FMs. On the other hand, Alhebaishi et al. [24] model and address cross-layer and co-residency attacks through VM placement optimization, focusing on a narrower range of attacks compared to our approach.

Also, there exist other works (e.g., [26, 27, 28, 29, 30, 31, 32, 4]) that verify security properties in virtual networks, e.g., clouds and SDN. Among them, ISOTOP [30] and Xu et al. [33] cover the consistency between different cloud layers. Additionally, there are other solutions, e.g., NetPlumber [26], Veriflow [27], and NoD [4] that verify flow rules against various security and functionality properties in virtual networks. However, none of these works considers NFV, and extending them to NFV would require significant efforts due to the added complexity. Table 2.1 compares existing solutions with NFVGuard+. It lists the solutions, whether they target NFV or other virtual environments, the NFV stack level they address, and the verified properties along with their verification methods. The symbols (✓) and (✗) mean supported and not supported, respectively.

Solution	NFV	Levels			Property	Method
		L1	L2	L3		
[21]	✓	✓	✗	✗	Security	Graph theoretic
[34]	✓	✗	✓	✗	Network	Custom algorithms
[16, 12, 17]	✓	✗	✗	✓	Correctness, performance	Trusted shim layer, graph theoretic
[7, 9, 11, 8, 18, 14, 15, 13, 19, 20], [10, 22]	✓	✗	✗	✗	Network, correctness, integrity	Remote attestation, MaxSAT solver, graph theoretic, custom algorithms, trusted shim, verified routing protocol, packet pair dispersion, tag-based verification, and machine learning
[29, 32, 26, 27, 31]	✗	✗	✗	✗	Security, operational, network, identity and access control	Graph theoretic, CSP solver, custom algorithms
NFVGuard+, [25]	✓	✓	✓	✓	Security and consistency	CSP solver, machine learning

Table 2.1: Comparing our solution with existing solutions.

2.2 Security Verification for NFV Using Machine Learning and Formal Method

Most existing solutions related to security verification for NFV (e.g., [16, 12, 35, 36, 14, 15, 13, 17]) focus on the verification of service function chaining (SFC). Those works employ either custom algorithms (such as [16, 15, 13]), graph-based methods (such as [12, 14, 17]), or formal methods (such as [35, 36]). Unlike those existing works (which focus on the SFC only), our previous work, NFVGuard [6], aims to verify the entire NFV stack (including both SFC and underlying infrastructure, and their consistency) using formal method. However, the increased scope also leads to increased complexity and longer verification time, which has motivated us to propose MLFM.

Besides NFV, there also exist security verification solutions for other virtual infrastructures, such as cloud and SDN (e.g., [26, 4, 37, 38, 31, 39, 29]), including formal method-based ones [4, 37, 38, 31]. Unlike MLFM, most such solutions do not specifically address the delay in verification (so they may benefit from MLFM in that aspect), with the exception of NOD [4] which is optimized for large applications (our experiments in Section 5.6 show it can further benefit from MLFM). In contrast to formal method, custom algorithms (e.g., [26] and [29]) may enjoy improved efficiency for specific properties but they generally lack the level of expressiveness of formal method-based approaches (including MLFM). Also designed to reduce verification time, the proactive approach (e.g., [40, 41]) performs the verification in advance based on predicted events, which is parallel to, and can be integrated with, our approach.

There exist works that combine machine learning and formal method in other contexts, such as automated program verification (for synthesizing invariants used to verify the correctness of a program, e.g., [42, 43, 44, 45]). In particular, Ezudheen et al. [42] develop learning-based algorithms for synthesizing invariants for programs that generate Horn-style proof constraints. Garg et al. [43] propose the ICE-learning framework for not only taking (counter-)examples but also handling implications. Ren et al. [44] propose a method based on selective samples to improve the efficiency of invariant synthesizing. Finally, Vizel et al. [45] study the relationship between SAT-based Model Checking (SAT-MC) and Machine Learning-based Invariant Synthesis (MLIS). Although the goals are very different (efficient verification vs. invariant synthesizing), our teacher-learner approach is similar to those existing works, with a key difference being that we additionally employ the sampling strategies from the active learning literature [46] to more effectively identify representative samples.

2.3 Security Verification for Microservices Using Federated Learning-Guided Formal Method

Existing security solutions for the microservice architecture mostly focus on verifying the correctness of the microservices interactions (e.g., [47, 48, 49]), anomaly detection (e.g., [50, 51]), access control for microservices (e.g., [52, 53, 54, 55]), or performance analysis (e.g., [56]). Those works employ either formal methods (such as [47, 48, 49]), graph-based methods (such as [50, 52, 56, 55]), ML methods (such as [51]), sidecar-based methods (such as [53]), or token-based methods (such as [54]). In contrast to those existing solutions, we tackle two unique challenges in the microservice architecture, i.e., the inherent complexity of formal security verification solutions may prevent them from handling the sheer scale of microservice applications, which can cause considerable delay in identifying security violations, and collecting data from all the microservice applications to perform security verification at a central location may be infeasible due to data confidentiality and privacy concerns.

Federated learning has been extensively leveraged in many security applications, such as anomaly (e.g., [57, 58, 59]) and intrusion detection (e.g., [60, 61]), mostly within the Internet of Things (IoT) architectures. To the best of our knowledge, there do not exist solutions that employ federated learning for compliance verification (directly or indirectly), nor does any work combine federated learning with formal methods for faster verification as we propose in this work.

Security verification solutions have also been developed for various virtual infrastructures, such as NFV, cloud, and SDN (e.g., [4, 35, 6, 36, 5, 37, 38, 31, 39, 40, 41]). Most of the FM-based solutions (e.g., [4, 35, 6, 36, 37, 38, 31, 39]) do not explicitly tackle the delay in verification (so they may benefit from FLFM in this regard), except for NOD [4], which is specifically optimized for large applications (NOD can potentially benefit from our approach, as shown in [5]), and MLFM [5], which is less efficient than our work, as shown through experiments in Section 5.6. Unlike formal methods, custom algorithms (e.g., [26] and [29]) benefit from enhanced efficiency for specific properties; however, they often lack the expressiveness found in formal method-based approaches. The proactive approach (e.g., [40, 41]) optimizes the verification time by performing the verification in advance based on predicted events, which is parallel to and can be integrated with our approach.

Table 2.2 summarizes the comparison between existing solutions and FLFM. The first and second columns enlist existing works and their objectives. The next two columns compare their applications and verification methods. The following two columns compare the coverage in terms of whether they work on a single microservice (MS-level) or they can support multiple microservice applications (App-level) as addressed in our work. The last column enlists the type of utilized federated learning (horizontal or vertical), if applicable. The symbols (✓), (✗), and (N/A) mean supported, not supported, and not applicable respectively.

Solution	Objective	Application	Method	Coverage		FL type
				MS-level	App-level	
Malchain [50]	Anomaly detection	Microservice-based cloud applications	Graph theoretic and ML	✗	✓	N/A
ucheck [47]	Correctness verification	Microservice-based applications	Formal method	✓	✗	N/A
AUTOARMOR [52]	Inter-service access control policy generation	Microservice-based cloud applications	Graph theoretic	✗	✓	N/A
Meng et al.[48]	Correctness verification	Microservice-based cloud applications	Formal method	✗	✓	N/A
Meadows et al.[53]	Anomaly detection and access control	Microservice-based applications	Secure sidecar	✗	✓	N/A
Venčkauskas et al. [54]	Access control	Microservice-based applications	Token-based	✓	✗	N/A
Dai et al. [49]	Interaction correctness of a microservice system	Microservice systems	Model checking	✗	✓	N/A
Zhang, et al. [56]	Performance analysis and anomaly detection	Microservice architectures	Graph theoretic	✗	✓	N/A
Pahl et al. [51]	Anomaly detection	Microservices-based Internet of Things (IoT) system	ML	✓	✗	N/A
Mothukuri et al. [57]	Anomaly detection	Internet of Things (IoT)	Federated learning	N/A	N/A	Horizontal
MV-FLID [60]	Intrusion detection	Internet of Things (IoT)	Federated learning	N/A	N/A	Horizontal
Liu et al. [59]	Anomaly detection	Internet of Things (IoT)	Federated learning	N/A	N/A	Horizontal
NOD [4]	Network verification	Cloud deployment	SMT Solver	N/A	N/A	N/A
NFVGuard [6]	Security and consistency verification	Network Functions Virtualization	Formal method	N/A	N/A	N/A
MLFM	Security verification	Network Functions Virtualization	ML and Formal method	N/A	N/A	N/A
FLFM	Security verification	Microservice-based cloud applications	Federated learning and formal method	✓	✓	Horizontal & vertical

Table 2.2: Comparing our solution with existing solutions.

Chapter 3

Cross-Level Security Verification System for Network Functions Virtualization

3.1 Introduction

The adoption rate of NFV is increasing[1] due to the many benefits of virtualizing proprietary physical devices in the network architecture, such as the capability for operators to scale their network services on-demand, and the lower cost of using existing cloud infrastructure. However, to attain such benefits, NFV involves multiple levels of virtualization and operates the managerial components at each level autonomously [63]. As a ramification, this additional complexity opens the door to potential inconsistencies among different levels of the NFV stack, which can be exploited to conduct stealthy attacks, e.g., “invisible” (to end users) security breaches at lower levels of an NFV stack [1]. To tackle such threats, verifying the security across different levels of an NFV stack (a.k.a. cross-level security verification) becomes essential.

To that end, most existing works (e.g., [7, 8, 9, 11, 13, 15, 16, 12, 17, 14, 18, 19, 20, 21, 22]) are insufficient as they typically focus on one particular level of the NFV stack,

[1] 92% of carriers have either deployed or plan to deploy network functions virtualization soon [62]

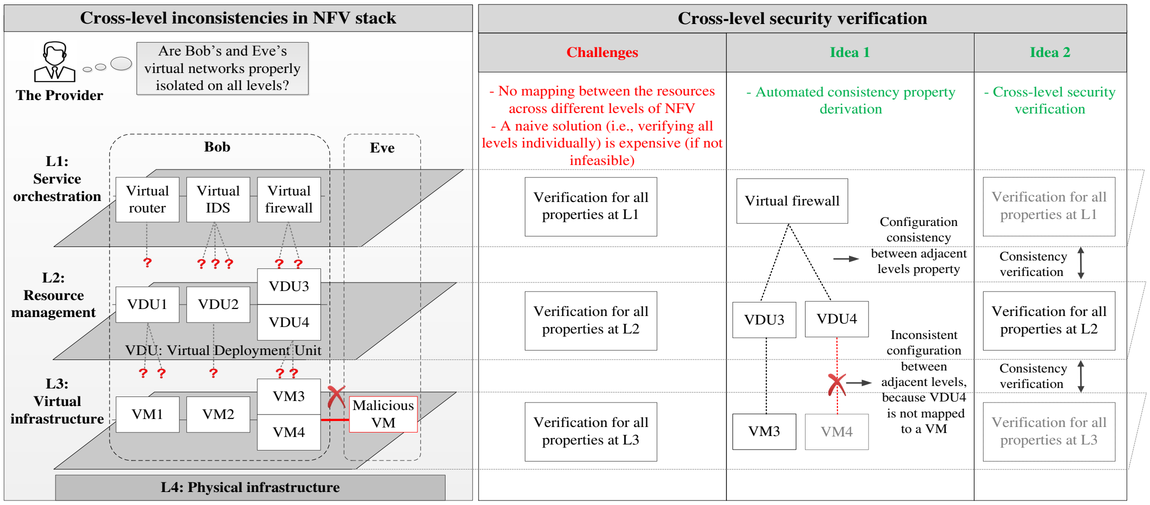

Cross-level inconsistencies in NFV stack
The Provider
Are Bob's and Eve's virtual networks properly isolated on all levels?
Bob
Eve
L1: Service orchestration
Virtual router
Virtual IDS
Virtual firewall
? ? ? ? ? ?
L2: Resource management
VDU1
VDU2
VDU3
VDU4
VDU: Virtual Deployment Unit
? ? ? ? ?
L3: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

Figure 3.1: A motivating example illustrating the challenges of cross-level security verification in NFV and our ideas.

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

Motivating Example. The left side of Figure 3.1 shows a simplified view of the NFV stack (Sec 2.1 provides more background on NFV stack) of two tenants, *Bob* and *Eve* (as indicated by the two dashed line boxes), which involve four levels (L1-L4 as indicated by the shaded planes) and their corresponding virtual and physical resources. We assume that, by exploiting real-world vulnerabilities (e.g., CVE-2024-1085 [64], CVE-2024-0193 [65], or CVE-2024-0646 [66]) in a specific way [1], a malicious tenant (*Eve*) could inject a malicious virtual machine, `Malicious VM`, into *Bob*'s network to secretly inspect his traffic at L3, without causing any detectable changes in the upper levels. Knowing about such potential threats, the provider is concerned with the following question: *"Are Bob's and Eve's virtual networks properly isolated at all levels?"*

The first column on the right side of the figure shows the existing challenges in cross-level security verification as follows. First, the mapping between the resources across different levels of the NFV stack (which might be useful for cross-level security verification) is unknown. Second, a naive solution which separately conducts security verification at each level of NFV through utilizing (multiple) existing works (e.g., [12, 14, 19, 20]) is expensive or even infeasible (e.g., its not always possible to re-define an L1 property at L4 in a meaningful way). To address those challenges, our two main ideas are illustrated in the next two columns of the figure. Specifically, our first idea is to identify the mapping between the resources in different levels of NFV and automatically identify the corresponding consistency properties (needed for the next idea). Our second idea is to only verify every property at the level where its specified (e.g., L2 in this case), and then implicitly extend the result of such verification to other levels by verifying the consistency between adjacent levels.

To instantiate those ideas, we propose a security verification system, NFVGuard+, for the efficient and practical cross-level security verification of NFV stack. NFVGuard+ leverages formal methods to model the audit data and properties as a Constraint Satisfaction Problem (CSP) and employs the Sugar solver [67] to verify compliance. To facilitate this, we first create an Entity-Relationship (ER) model to systematically capture NFV entities and their relationships. Next, we identify consistency properties from the ER model and design an algorithm to automatically derive them from the model. We then develop our cross-level security verification approach, utilizing the ER model for data collection, processing, and formal verification. Finally, we demonstrate the applicability of our solution by integrating it into a real NFV testbed based on OpenStack/Tacker [2] and evaluate its efficiency through experiments with both real and synthetic data.

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

Comparison to Existing Solutions. In comparison to most existing NFV security verification solutions (e.g. [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]), which primarily focus on a single level (mostly SFC), NFVGuard+ has a different focus, i.e., ensuring the security across all levels of the NFV stack. As we will demonstrate later in Section 3.2.3, this cannot be easily achieved using existing single-level solutions due to some unique challenges. Furthermore, although some approaches (e.g., [25, 24]) touch on the multilevel aspect of NFV, they do not formally model the verification problem as we do, cannot provide the same rigorous security proof provided by formal methods [25], or focus on a narrow scope of attacks (e.g., through VM placement optimization [24]). A detailed comparison is provided in Table 2.1.

3.2 Preliminaries

This section provides the preliminaries.

3.2.1 Background on NFV

NFV is a network architecture concept that virtualizes various network functions, such as routers, firewalls, load balancers, and intrusion detection systems (IDS) [3]. Figure 4.2 illustrates the multilevel NFV deployment model [1] (on the right) with the mapping to a simplified view of the ETSI NFV reference architecture [3] (on the left). The NFV deployment model complements the ETSI NFV reference architecture with deployment details found in multiple open source platforms including Networking Automation Platform (ONAP) [69], Tacker [70], OpenStack [2]. Specifically, the deployment model depicts the NFV stack at four abstraction levels: *Service Orchestration (L1)* (which supports the specification, on-boarding, and lifecycle management of network services. Also, it could optionally include the SDN Orchestrator (SDNO) for the automated management of network resources and services), *Resource Management (L2)* (which supports the instantiation of network services and the management of computing, storage, and network resources), *Virtual Infrastructure (L3)* (which hosts the virtual resources needed to support upper levels, and optionally the SDN controller (SDN-C)), and *Physical Infrastructure (L4)* (which includes all the physical resources).

3.2.2 Security Properties for NFV

Security properties of NFV define the desired security states of the NFV deployment that are usually specified by the tenants and/or providers. Very often, these properties are inspired by security standards (e.g., ETSI [63] and ISO 27002 [71]) that outline fundamental security principles and recommendations for guiding the providers and for assisting the

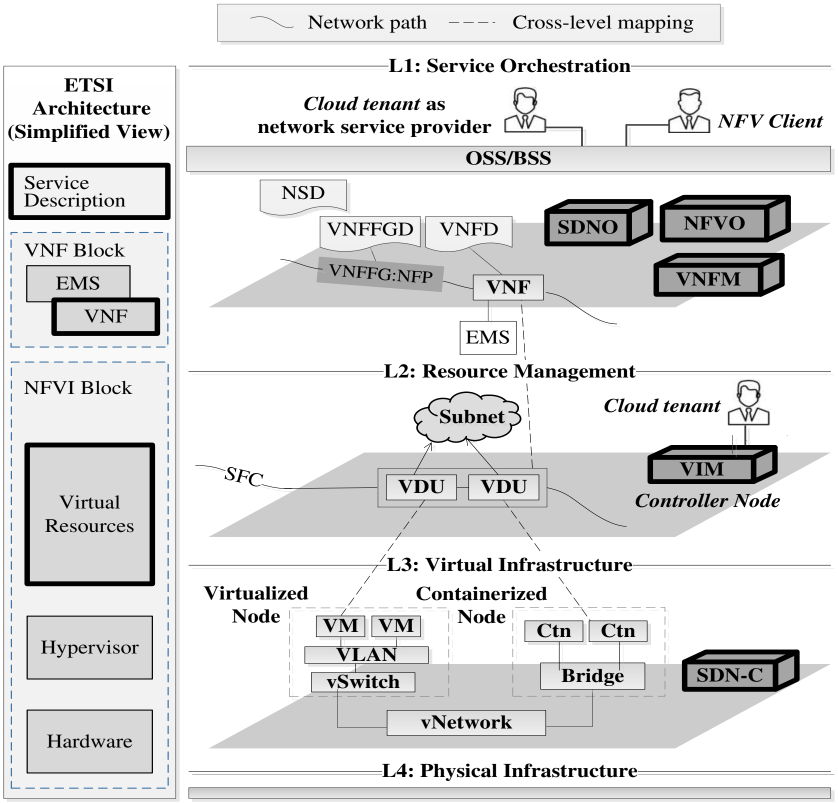


Figure 3.2: The multilevel NFV model [1].

tenants in assessing the overall security compliance with the provider's NFV infrastructure. For this purpose, we conduct a study on the standards related to NFV (e.g., IETF-RFC7498 [72], and ETSI [63]), along with the standards related to various components of an NFV stack, such as cloud and SDN (e.g., ISO 27002 [71] and CCM [73]). Then, we extract a list of security properties from those standards and the literature that can be used for the security verification of NFV. Table 3.1 shows extracted properties, their instantiation as sub-properties, descriptions of the sub-properties, and corresponding standards that require those properties for security compliance. Please note that while this list is not meant to be comprehensive, it can be easily extended to encompass additional security properties and even user-defined properties. Our approach can verify any security property as long as its expressed using formal methods. However, in this work we focus on verifying the compliance of security properties related to the static configuration of the virtualized infrastructure, such as the proper configuration of isolation mechanisms. Dynamic properties, such as those related to reachability and network forwarding functionality, are beyond the scope of this work and will be addressed in future work. To add specificity to our discussions, we provide a sample property, and subsequently, in Section 3.5, we show its verification process.

Example 3.2.1 Virtual resources isolation (no common ownership) property. Aims at verifying that each virtual resource is exclusively owned by a single tenant unless specified by a user-defined policy. Specifically, in this work we aim to verify that all VDUs composing a specific SFC at the management level are owned by a unique tenant, namely the owner of the SFC service.

3.2.3 Challenges to Cross-Level Security Verification

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

Identifying the NFV-Stack Entities and Their Relationships. To develop a cross-level verification system and identify the necessary input data for verifying various security properties, its essential to thoroughly understand the NFV system's design and workflow, which might be intractable as the NFV stack is a complex system with many inter-dependent entities located at different abstraction levels (as explained in Section 3.2.1). Moreover, the NFV standards (e.g., IETF-RFC7498 [72] and ETSI [63]) do not provide the necessary details for fully understanding the NFV system workflow and mapping the states of network services across the layers.

Security Properties	Sub-Properties	Description	Standards
Physical resource isolation [37]	No VNFs co-residence	VNFs of a tenant should not be placed on the same server as VNFs of a non-trusted tenant	ISO [74], NIST800 [75], CCM [73], ETSI [63]
Virtual resource isolation [37]	No common ownership	Tenant-specific resources should belong to a unique tenant, unless permitted by a user-defined policy	CCM [73], ETSI [63], IETF-RFC7665, RFC-7498 [72]
Topology isolation [37]	Mapping unicity VLANs-VXLANs	VLANs and VXLANs should be mapped one-to-one on a given server	ISO [74], NIST800 [75], CCM [73], ETSI [63], IETF-RFC7665, RFC-7498 [72]
	Correct association Ports-Virtual Networks	VNFs should be attached to the virtual networks they are connected to through the right ports	
	Overlay tunnels isolation	In each VTEP end, VNFs are associated with their physical location (at L2) and to the VXLAN assigned to the networks they are attached to at L1	
	Mappings unicity Virtual Networks Segments	Virtual networks and segments should be mapped one-to-one	
	Mappings unicity Ports-VLANs	Ports should be mapped to unique VLANs	
	Mappings unicity Ports-Segments	vPorts should be mapped to unique segments	
Policy and state correctness [76]	-	A policy can be dynamically changing. The changed policy should be reconfigured in VNF node as soon as possible	ETSI [63, 77], IETF-RFC7665, RFC8459[72]
Functionality of VNF and VNFFGs [16, 13]	-	Check if VNFs and the composition (i.e., service chaining) of these functions work as intended	ETSI [77], IETF-RFC-7665, RFC8459 [72]
SFC ordering and sequencing as defined by the specification [12]	-	SFCs should maintain the order of VNFs with the correct traffic forwarding behavior as defined by the specifications	ETSI [63, 77], IETF-RFC7665, RFC8459 [72]
Topology consistency [37]	VNFFG configuration consistency between L1/L2	Consistency between the size of VNFFGs, the sequences of VNFs and the classifiers at L1 and their parallel implementation at L2	ISO [74], NIST800 [75], CCM [73], IETF-RFC-8459 [72], ETSI [63, 77]
	Virtual links consistency	VNFs should be connected to the VLANs and VXLANs in L2 that corresponds to the virtual networks they are connected to in L1	
	VNF location consistency	Consistency between VNFs locations at L2 and L1	
	CPs-Ports consistency	Consistency between CPs defined at L1 and their created counterparts; Ports in (L2)	

Table 3.1: Examples of NFV security properties [6]

Locating the Data Sources for Security Properties. To verify a given security property, it's necessary to identify all relevant data sources and determine what data to collect from each source. This would require a good understanding of the property and accurately mapping its semantics to the corresponding NFV system resources, which also requires adequate awareness of the entities and their relationships within the NFV stack. Several security properties may require data from multiple levels of the NFV stack, depending on the involved data sources and their associated relationships. E.g., verifying SFC traffic isolation property [34], entails collecting data from the VDUs at L2, as well as from VMs and vSwitches at L3.

Data Correlation and Aggregation. Data sources are typically scattered across multiple physical servers and different NFV stack levels, each with its own data format (e.g., SFC traffic steering data is stored as OpenFlow rules at L3 and as database instances at L2).

Therefore, its necessary to process the data into a consistent format and piece together related data within the same level (i.e., data aggregation), especially when audit data is scattered across different tables (e.g., the SFC data resides in different Neutron tables and Nova databases). Moreover, we need to link between data across different levels (i.e., data correlation) to obtain the necessary information for verification. These challenges will be addressed in Sections 3.4 and 3.5.

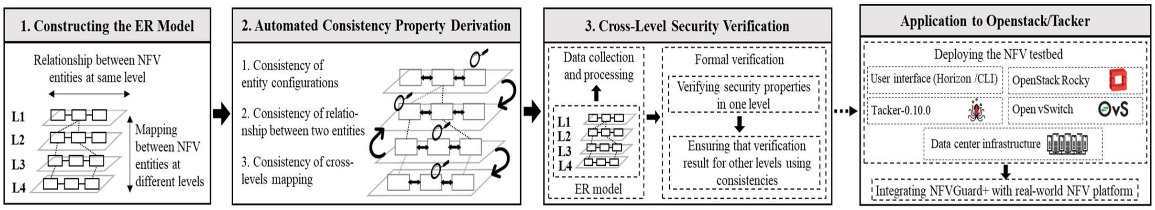

1. Constructing the ER Model
Relationship between NFV entities at same level
L1
L2
L3
L4
Mapping between NFV entities at different levels
2. Automated Consistency Property Derivation
1. Consistency of entity configurations
2. Consistency of relationship between two entities
3. Consistency of cross-levels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

Figure 3.3: An overview of the NFVGuard+ approach.

3.2.4 Threat Model

Our in-scope threats include both external attackers who exploit existing vulnerabilities in the NFV stack, and insiders such as cloud users and tenant administrators who cause security breaches either by mistakes or with malicious intents. Similar to most security verification solutions (e.g., [71, 29]), we trust the NFV provider for the integrity of the audit input data (e.g., logs and configurations). We also assume that the ER model correctly captures all the relations between the NFV system entities within the same level and captures all the mapping between cross-level entities, and any new changes in the system design affecting those relationships and mappings will be updated in the ER model. We assume that the properties defined in the work are correct and complete i.e., it encompasses all the data and required relations to describe the given property. We also assume that one-level security property verification combined with verifying consistency properties for all levels would be sufficient for cross-level verification of a security property (as detailed in Section 3.4). Whereas, consistency property inspects whether the specifications set by the tenants or service providers are implemented correctly in the NFV system and that the implementation of resources at a specific level is instantiated correctly at the underlying level(s). This work focuses on the verification of consistency properties and security properties related to the static configuration of the virtualized infrastructure, such as the proper configuration of isolation mechanisms. Any property violation that is not reflected on logs and configurations is beyond the scope of this work. Although dynamic properties, such as reachability-related properties, also can be verified through formal methods (Lopes et al. [4]), these are out of the scope and they will be investigated in our future work.

Additionally, although our cross-level security verification solution can detect a violation of security properties, its not designed to attribute such a violation to underlying vulnerabilities (i.e., vulnerability analysis) or specific attacks (i.e., intrusion detection). However, mitigation solutions (e.g., [78, 79]) can be applied to address the risks associated with security breaches or vulnerabilities. These include security hardening options such as updating and patching vulnerabilities, enforcing strict security policies and access controls, conducting regular security audits, penetration testing, hypervisor introspection, remote attestation, and rollback to known good configuration.

3.3 Overview

Figure 3.3 shows an overview of NFVGuard+ including its three major steps and application to NFV.

1. Constructing the ER Model. To model the interconnectivity between different components in an NFV system, we construct the ER model that mainly captures: (i) the relationship between NFV entities within the same level, and (ii) the mapping between NFV entities from different levels (detailed in Section 3.4.1).

2. Automated Consistency Property Derivation. We automatically derive consistencies (which will be used in cross-level security verification later) between different entities in the NFV stack based on the ER model. More specifically, we derive properties that include the: (i) consistency of entity configurations, (ii) consistency of the relationship between two entities, and (iii) consistency of cross-level mapping (detailed in Section 3.4.2).

3. Cross-level Security Verification. We conduct cross-level security verification by utilizing two major steps: (i) verifying a security property for one level, and (ii) applying that verification result to other levels using the consistency results. We also provide a general guideline for the users to identify new properties (detailed in Section 3.5).

Application to Openstack/Tacker. As a potential application of our solution, we integrate NFVGuard+ with OpenStack/Tacker (a popular choice for NFV deployment) [2]. In our implementation, the user-defined network service descriptors are uploaded to Tacker through Horizon/CLI [2]. We choose the latest version of OpenStack (i.e., Rocky) and Tacker (i.e., Tacker-0.10.0) [2] to obtain the most recent features of NFV deployments. Finally, the traffic steering among the VNF elements is handled by the OvS switches [80]. We will detail the testbed data generation approach, report implementation challenges, and describe the integration of NFVGuard+ into the testbed in Section 3.6.

3.4 ER Model Construction and Consistency Property Identification

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

3.4.1 Constructing the Entity Relationship (ER) Model

To capture the relationships between NFV entities (e.g., between *VNFFG* and *Path* entities) both within and across NFV levels, we devise an ER model for the NFV stack (shown in Figure 3.4). The shaded nodes represent NFV-related entities, while non-shaded nodes represent entities related to the underlying infrastructure. The directed edges show the relationships between those entities at the same level, while (1:1), (1:M), (M:1), and (M:M) represent the corresponding cardinalities of the relations. The dashed line edges represent the cross-level mapping between the entities at different levels, which have a (1:1) cardinality.

We construct the model by performing a comprehensive study of the system configurations of a real NFV testbed implemented using OpenStack/Tacker [2] (detailed in Section 3.6), and relevant literature on modeling and deploying NFV and virtualized infrastructures (e.g., [1, 30, 38]). We further validate our model with several industrial experts on NFV from a large telecommunication vendor. In the following, we elaborate on our ER model construction process.

Constructing the Nodes of the ER Model. According to the NFV deployment model (discussed in Section 3.2.1), we divide the ER model into four levels, the Service orchestration level (L1), resource management level (L2), virtual infrastructure level (L3), and physical infrastructure level (L4). Then, to capture the system entities at each level, we study the deployment details of NFV environments [2, 1] and the supporting technologies (such as network virtualization technologies like VLAN and VXLAN [81]) for implementing the NFV. Then, we represent the identified entities as the ER model nodes.

Example 3.4.1 In this example, we identify the `NS provider`, `NSD`, and `NS` nodes in the ER model. The NS provider uploads the Network Service Descriptors (NSDs) at the Service Orchestration level (L1), which define the network service based on user requirements. Each NSD creates one or more NSs, stored as entities in the NFV system, along with the NS provider's ID and information. Thus, the NS provider, NSD, and NS entities are represented as nodes in the ER model, as shown in Figure 3.4 at L1.

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

Example 3.4.2 Since the NSD creates the NS (as explained in Example 2), we establish a directed edge between these entities, labeled *CreatedFrom(M:1)*(Figure 3.4 at L1), to represent their relationship. The cardinality (*(M:1)*) reflects the constraints governing this relationship: the NSD can create multiple NSs, but each created NS belongs to only one NSD template. Furthermore, VNF specifications at L1 of the NFV system are instantiated as VDUs at L2. Thus, we represent this relationship as a cross-level mapping between the *VNF* and *VDU* entities.

3.4.2 Automated Consistency Property Derivation

This section illustrates how the ER model is utilized to automatically derive consistency properties (which will be used later for our cross-level verification in Section 3.5).

The relationships between entities in the ER model reflect fixed configuration constraints within the NFV system. For instance, the relationship between the *Path* and *Chain* entities at L1 (refer to Figure 3.4) is (1:1), indicating that each *Chain* is linked to a specific *Path*, and each *Path* corresponds to one *Chain*. Deviating from these fixed configurations can lead to unintended service behavior or interruptions.

Accordingly, we can derive properties, namely consistency properties, to verify whether

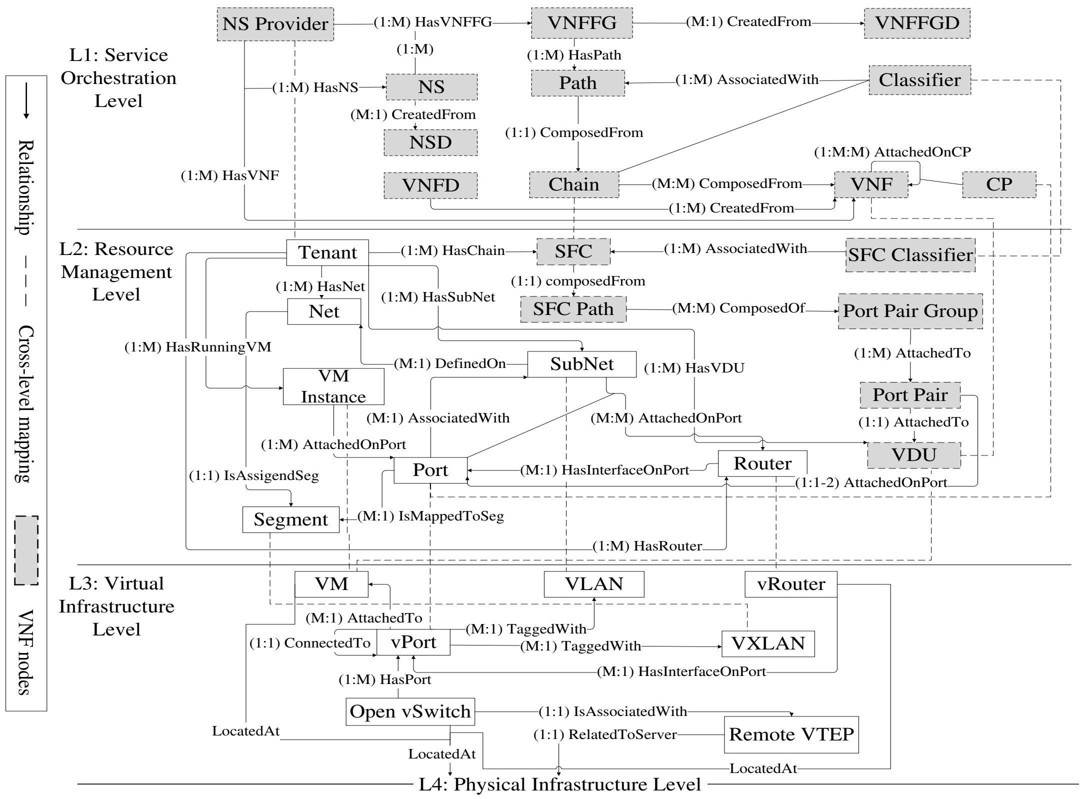


Figure 3.4: The ER model of the NFV stack.

any instances created within the NFV system comply with the established configurations. These properties can be automatically obtained by systematically parsing the entities and edges of the ER model, with the assumption that the model correctly captures all relationships between the NFV system entities at both the same and across different levels (Section 3.2.4). In particular, we can derive the following consistency properties.

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

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

Consistency of Relationships Between Entities Across Different Levels. Similarly, the dashed line edges between two entities in the ER model across adjacent levels represent a relationship between them. Specifically, this indicates that an entity at a higher level must have a corresponding implementation at the next level. We explore these edges to derive consistency properties that ensure the integrity of the mappings. Table 3.2 presents an excerpt of consistency properties automatically derived from the ER model, including their corresponding sources, and descriptions.

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

Table 3.2: Example of consistency properties identified from the ER model entities and relationships.

The aforementioned consistency properties can be automatically obtained from the ER model by representing it as a graph. In this graph, entities are depicted as nodes, and relationships are depicted as directed edges, attributed with relationships and their cardinalities. By traversing the graph, each node and its connected edges are processed to extract the relevant consistency properties. These properties are then stored in two lists: *EntityConsistencyProperty* for node-level consistency and *EdgeConsistencyProperty* for relationship-level consistency.

3.5 Cross-Level Security Verification

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

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

First of all, we must identify the property requirements (what needs to be verified) and determine their scope (which level they pertain to). Next, we will map these requirements at each level to the ER model and identify the entities within the model that relate to the property. For instance, the *VNFFG configuration consistency between L1/L2* property (refer to [6]) requires that the VNFFG design (at L1)-including the size of the VNFFG, the VNF sequences, and the classifiers definition-be correctly instantiated into corresponding SFC configurations (at L2), including the SFC size, VDU sequences, and classifier details. One of the requirements for this property at L1 is to determine the size of the VNFFG, which indicates the number of VNFs that comprise it. By referencing the ER model at L1, we should relate this requirement with the corresponding entities at this level. Since it pertains to the VNFFGs, we will select the *VNFFG* entity as a relevant entity for this property. Then, we will examine the relationships associated with this entity to check if they can be utilized by the property. For example, by examining the relationships associated with the *VNFFG* entity, we can observe that the *VNFFG* may consist of one or more paths, with each path comprising a chain of VNFs. This highlights the importance of the *Path*, *Chain*, and *VNF* entities in determining the size of the VNFFG.

Afterward, we will collect the relevant data for the property based on the identified entities and by consulting the data sources table (Table 3.3), which is created in conjunction with the ER model. For example, the ID of each VNFFG is stored in the *VNFFG* entity along with the ID(s) of the path(s) it is composed of, and each *Path* entity includes the ID of the chain that makes it up. While, the size of the VNFFG, can be determined from the data source of the `Chain` entity, as indicated in the *Description* column of Table 3.3. Likewise, the relevant data for the other requirements of the property are identified in the same manner.

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

Table 3.3: An excerpt of the data sources for some of the entities in the ER model, along with a description of the types of data they contain.

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

1. Data correlation: Due to the distributed nature of the audit data (e.g., data may be scattered across different services at the same level, such as Nova or Neutron in OpenStack or among physical servers), we need to correlate the collected data within each level to produce meaningful information for verification [30]. For example, *VNFFG_1* is implemented at L3 as three VMs (*VM_01*, *VM_02*, and *VM_03*) hosted on two physical servers. To verify the forwarding correctness of *VNFFG_1*, we need to collect the flow rules (determines how traffic flows through these VMs) stored on both physical servers and scattered across multiple tables on each server. For example, if *VM_02* and *VM_03* are on the same physical server and we want to verify whether *VM_02* is forwarding traffic to *VM_03*, we will need to examine the flow rules stored in tables 0, 5, 10, and the Group table. Therefore, we need to correlate all these data to piece together sufficient information for verification. The relationships between system entities at each level of the ER model help to identify the data that needs to be correlated. For instance, the relationship between the *VNFFG* entity and the *Path* entity indicates that they are interconnected and their data could be correlated.

2. Data aggregation: Audit data for specific properties, such as consistency properties, could be distributed across different levels of the NFV stack. Therefore, we must aggregate the data from these different levels to compile sufficient information for verification. For example, to verify whether a VNFFG is correctly implemented according to the specification, we need to collect the specification data at L1, aggregate it with the instantiation data at L2, and further combine it with the implementation data at L3. The cross-level mapping relationships between system entities at each

level of the ER model assist in identifying the data that needs to be aggregated.

Formal Verification. We propose to apply formal methods to verify the compliance of the NFV stack against the identified security and consistency properties. In this work, we formalize the properties as a Constraint Satisfaction Problem (CSP), a time-proven technique for expressing many complex problems. We then apply Sugar [67], a well-established constraint solver, to check whether these properties are satisfied. We detail the verification process as follows.

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

- Variable and domain declaration. Entities of the ER model—i.e., the nodes representing system components—are expressed as CSP variables with integer domains. Each domain encompasses all data instances defined within the system. For example, for the VNFFG configuration consistency between L1/L2 property, the VNFFG entity (refer to Figure 3.5) is expressed as the variable *fg* defined over the domain *VNFFG* such that `(domain VNFFG 0 max_vnffgs)` is a declaration of a CSP finite domain of VNFFGs, where each value between `0` and `max_vnffgs` is for a corresponding data instance in the NFV system.

- Relation declaration. The ER model relations, involved in the property requirements, are converted into CSP relations over variables with a support consisting of tuples of system data. For example, the relation between the VNFFG and its path (refer to Figure 3.5) is defined as the CSP relation `(relation HasPath 2 (supports(fg path))`, where instances of a given relation are the set of tuples corresponding to the entities instances. The CSP relations describe the current state of the system.

- Constraint declaration. We define constraints, in terms of CSP predicates, over the involved relation to specify the conditions that the instances of these relations should satisfy. Since CSP solvers provide solutions only in case the constraint is satisfied (SAT), we define constraints using the negative form of the property to obtain a counter-example in case of a violation.

- Body. We combine different predicates based on the properties to verify using Boolean operators.

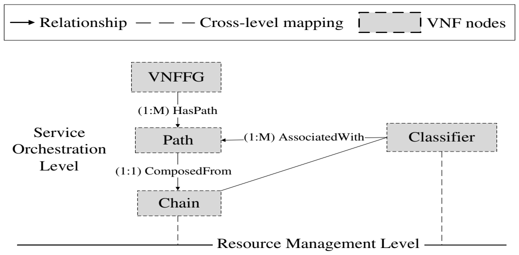


Figure 3.5: Thumbnail of the ER model showing entities for verifying VNFFG configuration consistency property at L1.

When the CSP solver (i.e., Sugar) solves the constraints and finds no solution (UNSAT), the verified properties are reported to be compliant. Otherwise, the solution provided by the CSP solver gives the variables' instances for which the negative form of the property is satisfied, meaning that a violation has occurred. For instance, we express the property virtual resource isolation presented in Example 3.2.1 using the following CSP relations. `HasChain(t, sfc)` which evaluates to true if tenant *t* has/owns a running SFC *sfc*, `SFCHasVDUs(sfc, vdu)` which evaluates to true if the SFC *sfc* has assigned VDU *vdu*, `HasVDU(t, vdu)` which evaluates to true if the tenant *t* has a running VDU *vdu*. Then we define the negation of the property in terms of a predicate over those relations to obtain a counter-example in case of a violation, shown as the `VirtualResourceIsolation` predicate in Listing 3.1 (an excerpt of Sugar code). Example 3.5.1 shows how Sugar verifies this property and allows for obtaining the violation evidence.

Example 3.5.1 Suppose that a tenant *t* with the `Tenant_ID` (18e552) is encoded as (10) in listing 3.1, the `Chain` (3cf7ca68) he owns as (1), and the VDUs (49ce0b1e, 738bb405) as (15, 16), respectively. The predicate `VirtualResourceIsolation` will evaluate to true if any of the VDUs assigned to the chain (1) that belongs to tenant (10) is owned by another tenant. According to the relation instance `HasVDU(11 16)`, the chain (1) has a VDU (16) that does not belong to tenant (10). Therefore, the predicate evaluates true and the output of Sugar code is (SAT) with evidence about what values breached the property i.e., (t=10; sfc=1; VDU1=16).

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

Listing 3.1: An excerpt of Sugar source code.

After verifying the NFV-related properties, we ensure the verification result for other levels using consistencies. The consistency between different levels of the NFV stack can be utilized to improve the performance of the verification (as we illustrate in the motivating example in Section 3.1). The key idea is to leverage the consistency result to perform security verification at one level of the NFV stack, instead of verifying the same security property at each level separately. As long as the NFV stack levels are consistent, the verification results at one level would be applicable to other levels. We show the performance improvement that we gain by utilizing the consistency property in experiments (Section 3.1).

3.6 Application to OpenStack/Tacker

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

3.6.1 Deploying the NFV Testbed

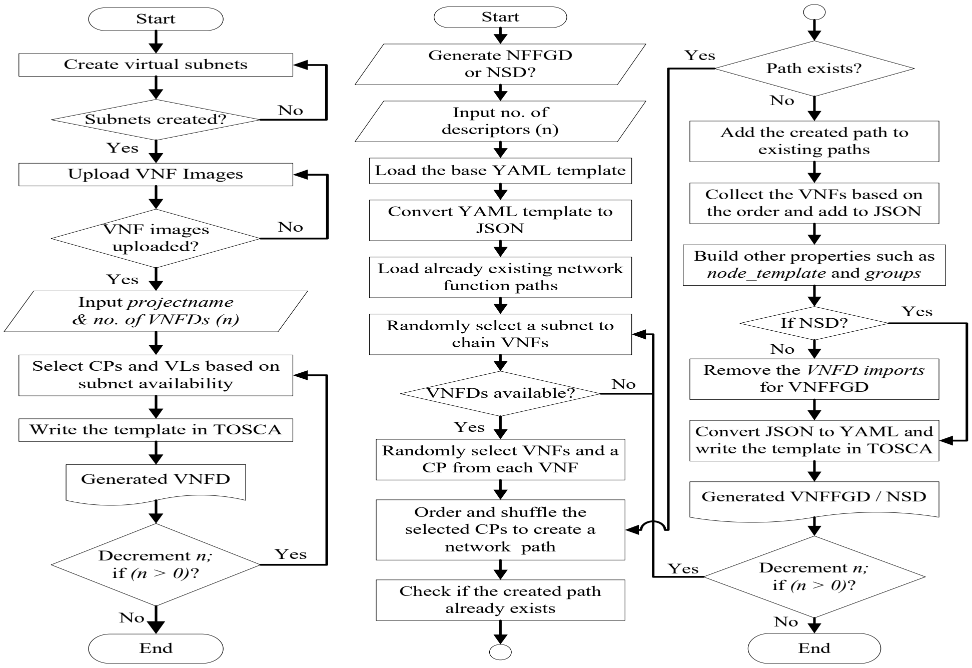
NFV Testbed Implementation. We build our NFV testbed using OpenStack [2] with Tacker [70] due to its growing popularity in the real world (e.g., [68]). More specifically, we rely on OpenStack for the Virtual Infrastructure Manager (VIM), which has been adopted by 96% of CSPs and more than 60% of the telecom operators in their NFV deployments [82]. We rely on Tacker, an official OpenStack project, for both VNFM and NFVO modules based on the ETSI MANO architectural framework [3]. We choose the latest version, i.e., OpenStack Rocky and Tacker-0.10.0 [2] to obtain the most recent features of NFV deployments.

NFV Data Generation. We intend to deploy a large-scale NFV system to assess the performance of NFVGuard+. However, to the best of our knowledge, there is no publicly available dataset of TOSCA [83] deployment descriptors for a large-scale NFV deployment. Therefore, we develop Python scripts to generate various Virtual Network Function

Descriptors (VNFDs) and Virtual Network Function Forwarding Graph Descriptors (VNFFGDs) in TOSCA, and we onboard those to our testbed to deploy different network services and generate large-scale NFV datasets. To ensure more diversity, we randomly choose a few parameters in the template while generating the deployment descriptors: 1) the number of network ports per VNF, 2) the number of VDUs per VNF, 3) the Flavor for each VNF and VDU, 4) the number of VNFs for each Network Function Path (NFP), 5) the order of VNFs for each NFP, 6) the flow-classifier criteria for each NFP, and 7) the number of NFPs for each VNFFG.

Specifically, the scripts first generate a diverse set of VNFDs for a given tenant by customizing a base template. After that, they generate multiple VNFFGDs (resp. NSDs) by creating unique network function paths using the available VNFDs. Then, these descriptors are onboarded to the VNFM and the NFVO modules in Tacker, respectively, through Horizon/CLI [2]. Once onboarded, the TOSCA templates are interpreted and translated to Heat templates [2]. Then, using the Heat template, Tacker leverages Nova to provision the virtual instances implementing the VNFs, and Neutron to provision the virtual networks that provide the connectivity to and from each VNF. Finally, the traffic steering among the chains of VNFs is handled by the OvS switches [80]. Figure 3.6 shows the detailed flowcharts for generating VNFDs and VNFFGDs.

VNFDs Generation. Figure 3.6(a) depicts the procedure to generate multiple VNFDs for a given tenant. Each VNFD is used to create one or several VNFs of the same type. More specifically, we use a base template that our generator customizes to create a diversified set of VNFDs. First, a set of virtual subnets is created, and then the corresponding VNF images are uploaded to be used within the VNFD templates. For each VNFD to be created, a set of subnets is selected, and then identifiers for connection points (CPs) and virtual links (VLs) associated with the VNF are created accordingly. Then, these identifiers are applied to fill in the VNFD template. Once these VNFDs are generated, they are used to generate


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

(a) VNFD Generation

(b) VNFFGD/NSD Generation

Figure 3.6: The process of generating VNF and VNFFG/NS TOSCA template descriptors.

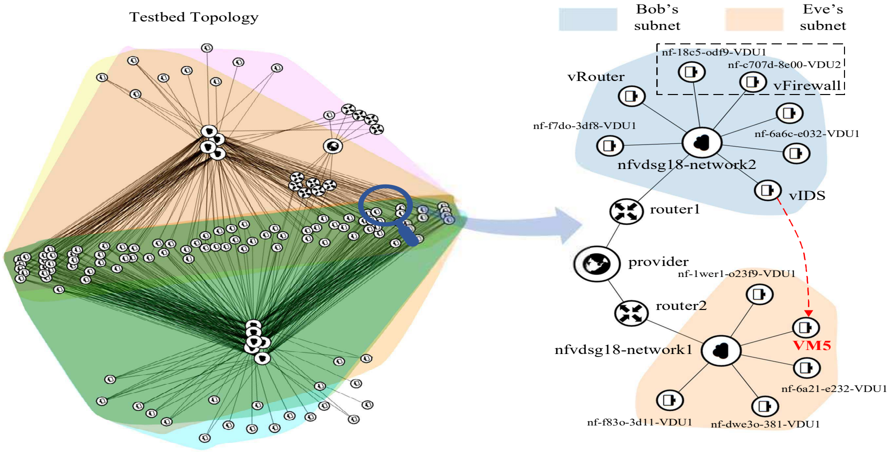


Figure 3.7: The topology of our NFV testbed (left) consisting of 20 tenants, 200 VNFFGs, and 200 VNFs and detailed view (in Horizon [2]) of an attack scenario similar to the motivating example in Section 3.1 (right).

VNFFGDs/NSDs.

VNFFGDs/NSDs Generation. Figure 3.6(b) depicts the process for generating multiple VNFFGDs (resp. NSDs) for a given tenant. Similar to generating VNFDs, we use a base

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

Figure 3.7 (left) is generated using OpenStack Horizon [2] to provide an overview of the network topology of our NFV testbed consisting of 20 tenants, 200 VNFFGs (each VNFFG consists of 10 VNFs), and each tenant has 10 VNFFGs. The figure shows the interconnections between the provider network and different tenant subnets (which are highlighted in different colors for each tenant) with their corresponding routers and VNFs. Figure 3.7 (right) shows a detailed view of an attack scenario similar to the motivating example (Section 3.1) where a malicious virtual machine (`VM5`) from the network of *Eve* (`nfvdsg18-network1`, highlighted in orange), is stealthily added to the service function chain of *Bob* implemented in his subnet (`nfvdsg18-network2`, highlighted in blue).

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

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

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

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

3.6.2 NFVGuard+ Implementation

The data collection component is implemented to collect data from different OpenStack services, such as Tacker, Nova, and Neutron [2], as well as from the instances running on every compute node of OvSs. Specifically, we rely on the Tacker database to retrieve user-defined descriptors uploaded to the VNFM and NFVO modules of Tacker (e.g., VNFD and VNFFGD) as the basis for verifying most of the properties. We also rely on a collection of OpenStack databases, such as Neutron database for information about SFC networking (e.g., the sequence of service functions, the traffic steering in-between, and the traffic classifier) and Nova databases (e.g., table *Instance*) for information about the tenant, the VDU, and the hosting machine. Finally, we collect the OpenFlow tables and internal OvS databases from all the compute nodes, e.g., to check for properties such as inconsistencies between L2 and L3. To process the collected data, we implement the data processing component in Python and Bash scripts. First, for each property, our processing component identifies the involved relations, and the supports of the relations are either fetched directly from the collected data (e.g., the support of the relation *BelongsTo*) or recovered after data correlation. Second, our processing component formats each group of data as an n-tuple, e.g., (resource, tenant), (OVS, VLAN, VXLAN), etc. Finally, it uses the n-tuples to generate part of the Sugar [67] source code and appends the n-tuples with the variable declarations, relationships, and predicates for each security property. Then we develop a customized script to generate the Sugar source code for the verification of each property. The formal verification component is implemented to feed the generated code into the Sugar CSP solver version 2.3.3 [67]. Sugar then produces the verification results to either state the property holds or provide evidence when the property is breached.

3.7 Experiments

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

3.7.1 Experiments with Synthetic Data

Experimental Settings. We deploy our testbed on a SuperServer 6029P-WTR equipped with Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz and 128GB of RAM. To evaluate the performance of NFVGuard+, we generate various synthetic datasets of different sizes varying from 1K up to 5K VNFFGs (representing reasonably large NFV setups [84]), and from 20K to 100K VMs. All data processing and experiments are conducted on the SuperServer with the verification tool, Sugar V2.3.3 [67]. Each experiment is performed 1,000 times to avoid any fluctuation caused by other operations on the server. The reported results show the efficiency and scalability of NFVGuard+.

Effectiveness Evaluation of NFVGuard+. To evaluate the effectiveness of our approach, we apply NFVGuard+ to pre-validated instances of security properties and assess its accuracy in verifying those instances. Table 3.4 shows some example security properties, their investigated instances, the instantiated Sugar code for each instance, and the corresponding Sugar output.

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

Table 3.4: Example property instances for evaluating the effectiveness of NFVGuard+.

The accuracy of our approach depends on the precision of the formal verifier, specifically the Sugar SAT solver. To test the solver's accuracy, we provide the solver with pre-validated instances and compare its results with our own. In particular, we verify instances of the *VNFFG configuration consistency*, *virtual resource isolation*, and *mapping unicity VLANs-VXLANs* properties. These instances are first tested by us for compliance before being given to the solver. Then, we check if the solver incorrectly identifies any of the compliant instances as non-compliant. Our evaluation shows that the solver output is accurate, correctly identifying all instances as compliant. Examples of these instances are shown in the first four rows of Table 3.4, where the solver output is UNSAT, indicating that the instances comply with the corresponding properties. For clarity, the instantiated Sugar code has been shortened and simplified. For more details on Sugar syntax and the full code excerpt, refer to Section 3.4.

Next, we inject security breaches at different levels of the NFV stack and test the accuracy of our approach in identifying those breaches. First, by exploiting a privilege escalation vulnerability in OpenStack (OSSA-2017-004 [85]), we would be able to modify the specification of an SFC and add an additional VNF. Such a modification at L2 will not be reflected at L1, resulting in a breach of *configuration consistency between L1/L2* property. An instance of this breach is presented in Table 3.4. In verifying the property, the solver aims to identify any VNFs that are defined in L1 for the given chain but not in L2, and vice versa. The solver successfully identifies this breach and returns the values that cause the property violation, specifically: (VNFFG path: 20fp, SFC: 20fp, VDU2: 11f, VDU3: 12f) (refer to Table 3.4).

Second, we target the flow tables at L3 to create inconsistencies with higher levels. By triggering a virtual switch reconciliation during a network topology update, outdated flow rules are reinstalled, causing traffic to be steered according to old definitions [1]. This leads to a breach in configuration consistency between L3 and upper levels. An instance of this breach is presented in Table 3.4, where we assume the *configuration consistency between L1/L2* property was verified to be met by the configuration.

In particular, a VNFFG that initially forwarded traffic from a vRtr to a vFW is updated to route traffic from the vRtr to the vFW and then to an additional vDPI. While the SFC at L2 is updated, the L3 flow rules remain unchanged due to the virtual switch reconciliation vulnerability. In verifying the property, the solver aims to identify any discrepancies in the traffic steering information collected from the different levels. The solver successfully identifies this breach and returns the values (SFC: 30fp, Chain: 30fp, vFW: 18f, vDPI: 19f) as evidence of the violation. Additionally, we generate misconfigurations to create further breach instances. The last two rows of Table 3.4 provide examples of these instances.

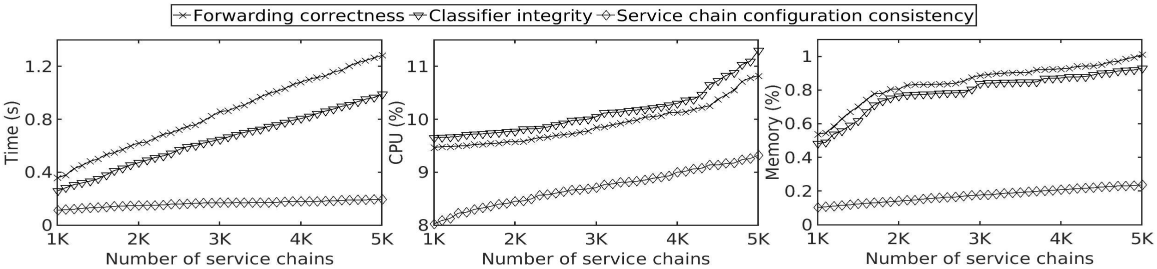


Figure 3.8: Verification performance for the consistency properties while varying the number of service chains.

Our tests demonstrate the effectiveness of our approach in providing accurate results for the specified security properties. In general, using formal methods in security verification is known to provide provably accurate results [86, 87] for given security properties. A practical challenge is for administrators to properly identify and define the security properties based on their specific needs. One potential solution to this challenge is to automatically extract security properties from standards using natural language processing (NLP) [88, 89], though this falls beyond the scope of this work.

Efficiency of Verifying the Consistency Properties. In this experiment, we evaluate the efficiency (in terms of response time, CPU usage, and memory consumption) of NFV-Guard+ in verifying the consistency properties derived from the ER model (refer to Section 3.4.2). We verify the *classifier integrity*, *forwarding correctness*, and *service chain configuration consistency* properties, which correspond to the consistency properties derived from the different objects of the ER model, i.e., node, edge, and cross-level edge, respectively.

According to Figure 3.8, the verification time increases almost linearly with the increased number of resources and the verification requires less than 1.5 seconds for all three properties even for the largest dataset. The verification of *service chain configuration consistency* property incurs the lowest response time, CPU, and memory consumption as shown in Figure 3.8 due to its simplest predicate with a smaller number of variables than the other two properties. This is expected as complex properties with a higher number of relations and variables generally take more time to process, and consume more memory

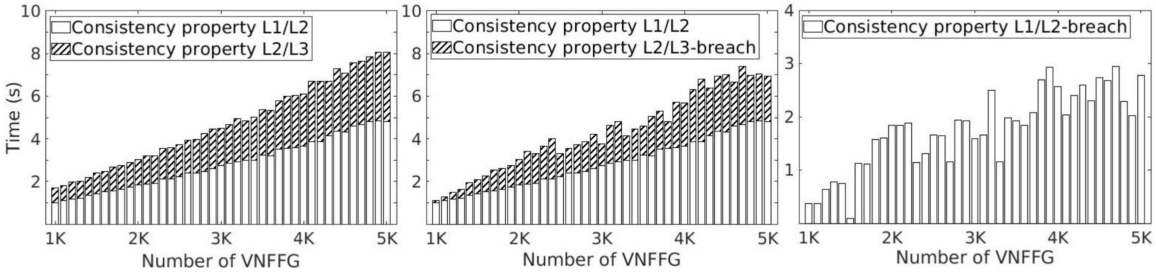


Figure 3.9: Verification time for the topology consistency properties in case of compliance (left), in case of reporting the first breach verifying between levels L2/L3 (middle), and in case of reporting the first breach verifying between L1/L2 (right).

and CPU. However, the maximum amount of CPU consumption is less than 12%, while the maximum memory consumption is only 1%. Hence, though the verification for the *forwarding correctness* property takes more time and consumes more resources than the *classifier integrity* property, the consumed resource still stays reasonably low.

Efficiency of Cross-Level Security Verification. In this set of experiments, we evaluate the verification time required by the candidate properties presented under different configuration scenarios. More specifically, the first configuration scenario assumes that the NFV configuration has no violation of any of the considered properties (detailed later in this section and depicted in Figures 3.9 (left) and 3.10 (left)), while in the second scenario (detailed later in this section), we inject several violating instances for each of the tested properties and consider the time to report the evidence only for the first breach (Figures 3.9 (middle) and (right) and 3.10 (middle)), in case a fast binary answer on the compliance status of the system is required by the system administrator/auditor. We then consider the average response time to find all compliance breaches (detailed later in this section and Figures 3.10 (right), 3.11 and 3.12 (left)). For each of the investigated scenarios, we consider the consistency properties, *VNFFG configuration consistency between L1 and L2*, and the *VNFFG configuration consistency between L2 and L3*. We also consider the security properties, *virtual resource isolation* (L2), and the *mapping unicity VLANs-VXLANs* (L3).

Note the required time for detecting non-compliance with the consistency properties also depends on the level where the breach is detected. For instance, if we detect a violation in the consistency property at L1/L2, then the verification stops and we report the time for non-compliance of the VNFFG configuration as the time for non-compliance of the later consistency property (Figures 3.9 (right) and 3.11 (middle)). Since the hierarchy of the NFV stack implies that any faulty configuration at the higher levels would lead to a fault at the lower levels, to reduce the verification time, we exploit this observation and stop the verification once we have a violation at higher levels. Otherwise, we continue the process to verify the non-compliance of the consistency property between lower levels (e.g., L2/L3). In this case, the verification time is the time for verifying the consistency property between L1/L2 in case of no breach and the time for reporting non-compliance at the lower levels L2/L3 (Figures 3.9 (middle) and 3.11 (left)).

Scenario 1. Cross-Level Security Verification in Case of Compliance: Figure 3.9 (left) depicts the verification time for the consistency properties in case of compliance. In general, the consistency property verification consumes more time for verifying between higher levels (it requires 1∼5s for L1/L2) than for lower levels (1∼3s for L2/L3) due to a more complex and higher number of relation instances of the predicates between higher levels. Moreover, we also observe that with an increased number of VNFFGs, the required time is increasing almost linearly.

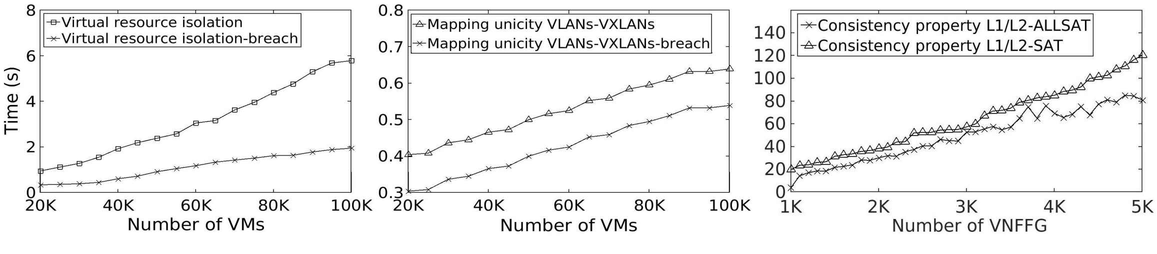


Figure 3.10: Verification time for the security properties virtual resource isolation (left) and mapping unicity VLANs-VXLANs (middle) in case of compliance and in case of reporting the first breach. Verification time for finding all compliance breaches (10 breaches) for the consistency property L1/L2 using SAT and ALLSAT solvers (right).

Scenario 2. Cross-Level Security Verification in Case of Detecting the First Breach:

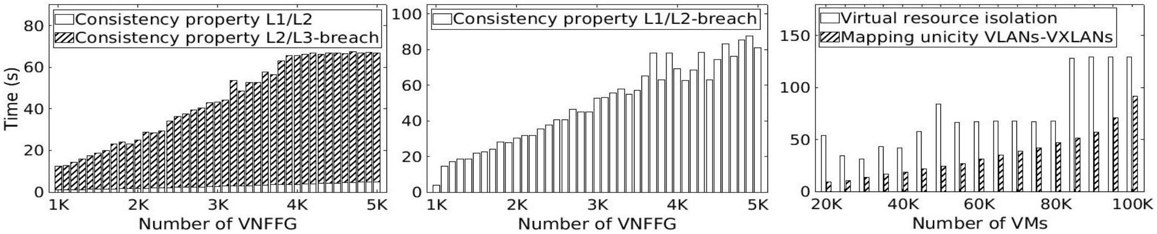


Figure 3.11: Verification time for the topology consistency properties, virtual resource isolation, and mapping unicity VLANs-VXLANs in case of reporting all compliance breaches using ALLSAT solver, with (left) reporting all breaches for verifying between levels L2/L3, (middle) reporting all breaches for verifying between L1/L2, and (right) reporting all breaches for verifying virtual resource isolation and mapping unicity VLANs-VXLANs.

Figure 3.9 (middle and right) depict the verification time for the consistency properties in case of non-compliance and providing the evidence for the first security breach. The time to detect and report the first breach ($\sim$ 3s while the first breach was found at L1/L2, and $\sim$ 6s for L2/L3) is less than the time required for assessing the same property in case of compliance ($\sim$ 8s). This is due to the action of immediately stopping the verification process after finding the first breach as we mentioned earlier. Also, the time for detecting breaches between lower levels is not far from the time in the case of compliance, which can be attributed to the fact that the verification of consistency property between higher levels is more time-consuming than the one between lower levels. We consider the time for detecting non-compliance to be reasonable for application in real life as a non-real-time auditing solution.

Figure 3.10 (left and middle) show the time for verifying the security properties in case of compliance and reporting the evidence for the first breach. The verification of mapping unicity VLAN-VXLAN is more efficient (less than 1 second), and the required time increases more slowly than it does for the virtual resource isolation (6 seconds for the largest dataset) as the latter has more complex predicates involving a higher number of relation instances. Similarly, as in the case of consistency properties, the time for reporting the breach is shorter than the time for asserting compliance for both of the security properties.

Scenario 3. Cross-Level Security Verification in Case of Detecting All the Breaches: Figure 3.10 (right) shows the average verification time to find all compliance breaches for the consistency property L1/L2 for both the case of using SAT [67] and ALLSAT [90] solvers, while the given number of breaches in each dataset is 10. The figure shows that the ALLSAT solver is faster than SAT solver in finding all the security property breaches. The reason is that SAT solvers can only provide a single solution in each run, while ALLSAT solvers are capable of finding multiple breaches in a single run. As a ramification, to find all the solutions, we have to run the SAT solver again and again until finding all breaches (i.e., for determining 10 breaches in the experiment, we have to run the solver 10 times). Hence, ALLSAT is clearly more applicable when finding an exhaustive list of breaches is desirable.

Consequently, we analyze the efficiency of the ALLSAT solver in detecting all 10 breaches (Figure 3.11). Figure 3.11 (left) shows the average time for detecting non-compliance breaches at the lower levels ($\sim$ 66s) for the largest dataset (5K VNFFGs), and Figure 3.11 (middle) shows the verification time for detecting non-compliance breaches at the higher levels ($\sim$ 80s) for the same dataset. The ALLSAT solver could efficiently find all the violations, and the time for detecting multiple violations is longer than the time of detecting a single solution and assessing the compliance of the system in case of no breach. This indicates that the verification time of our solution increases with an increasing number of violations. Also, the time for finding all the breaches for the consistency property at L1/L2 is more than that for the consistency property between L2/L3, which is related to the complexity of the property at the higher levels. Therefore, the verification time for the lower levels is less than that of the higher levels, especially that the time for compliance verification of the consistency property at L1/L2 is as short as $\sim$ 5s. Moreover, even though the verification of the higher levels takes more time, its still much faster than the naive approach of verifying both of the properties, which would take both the time for verifying the

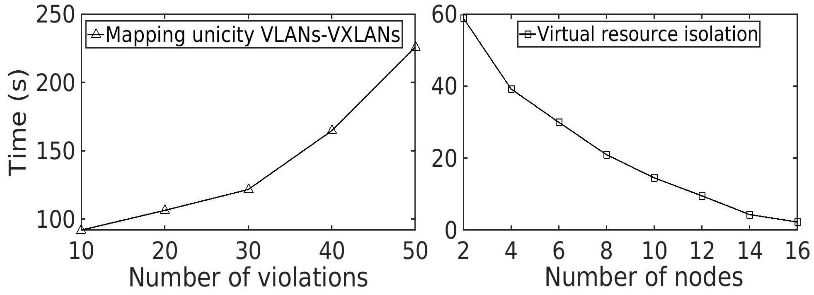


Figure 3.12: Verification time for reporting all breaches for the security property mapping unicity VLANs-VXLANs while varying the number of breaches (left) and the time for parallelizing the verification of the virtual resource isolation property (right).

consistency property between L1/L2 in case of non-compliance (i.e., $\sim$ 80s) and the time for verifying the consistency property between L2/L3 in case of non-compliance (i.e., $\sim$ 62s).

Figure 3.11 (right) shows the time for verifying the security properties in case of reporting all compliance breaches. The time of reporting all compliance breaches for both of the security properties is longer than the time for reporting compliance. Moreover, the time for reporting all breaches of the *virtual resource isolation* property ($\sim$ 2m for the largest dataset (100K VMs)) is higher than the time for reporting all breaches of the *mapping unicity VLANs-VXLANs* property, as the latter is more complex. Figure 3.12 (left) studies the effect of increasing the number of violations on the verification time. In this experiment, we verify the *mapping unicity VLANs-VXLANs* property, and we vary the number of violations encountered in the dataset where the dataset size is (100K VMs). As depicted in the figure, the time increases almost linearly with the number of violations, and it takes about 3.7m to verify 50 breaches.

Efficiency Improvement Due to the ER Model. This set of experiments is to evaluate the efficiency improvement (Figure 3.13) resulting from utilizing the ER model in multi-level security verification by comparing its required time with that of a conventional security verification approach (i.e., conducting security verification at each level). We verify the "SFC ordering and sequencing as defined by the specification" security property (defined

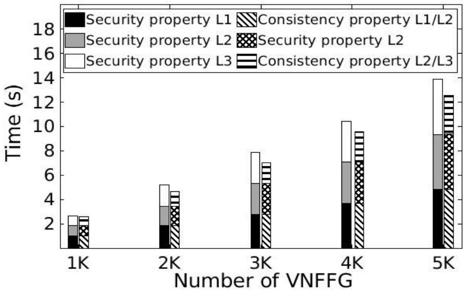


Figure 3.13: Comparing the verification time of the multi-level security property without (the grayscale bar) and with (the bar with patterns) the utilization of ER model.

in [6]), which checks if the deployed SFCs maintain the order of VNFs with the correct traffic forwarding behavior as defined by the specifications.

Figure 3.13 shows the required time for the multi-level verification for the "*SFC ordering and sequencing as defined by the specification*" security property. The grayscale bar represents verifying this property at each level of the NFV stack as mentioned in the motivating example (Section 3.1). The bars with patterns show the required time for verifying the same property with the existence of the ER model i.e., by verifying the consistency between the NFV stack levels after verifying the security property at one level (i.e., L2 in the figure, the middlebox with solid gray color). Each bar in the figure consists of three portions, where each portion represents the required time for verifying the security properties at each level or the consistency properties. With the help of the ER model, its possible to only conduct the security verification at one level (e.g., L2) and then conduct the consistency verification for the adjacent levels. Figure 3.13 depicts that the implementation of the ER model reduces the verification time; for instance, for the largest dataset (5K SFCs), the implementation of the ER model reduces the overall verification time by 1.4 seconds.

Applicability of NFVGuard+ to Different Solvers. The intention of this experiment is to investigate the applicability of NFVGuard+ to different SAT solvers. Our implementation is based on Sugar, which is an SAT-based constraint solver, where the CSP is solved by a backend SAT solver. Sugar supports MiniSat [67] as the default backend SAT solver.

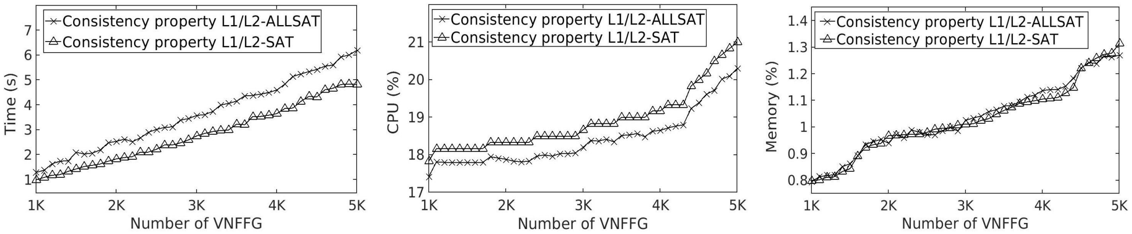


Figure 3.14: Verification performance for the consistency property L1/L2 using ALLSAT and SAT solvers.

Our next experiment also investigates ALLSAT (i.e., short for all solutions SAT) backend solver [91], a variant of SAT solvers that deals with enumerating all satisfying assignments of a propositional logic formula. To the best of our knowledge, clasp [92], PicoSAT [90], and relsat [93] are the only ALLSAT solvers. Since PicoSAT is the only ALLSAT solver supported by Sugar, we consider this in our implementation.

More specifically, to demonstrate the applicability of NFVGuard+ to different solvers, we implement Sugar to assess the consistency properties at L1/L2 using ALLSAT (i.e., PicoSAT) and SAT (i.e., MiniSat) solvers. Figure 3.14 illustrates this verification performance in terms of time, CPU, and memory. Figure 3.14 depicts that the performance of both ALLSAT and SAT solvers is mostly similar. Generally, for both these solvers, resource consumption increases almost linearly with the increased number of VNFFGs. The ALLSAT solver requires a slightly longer verification time (Figure 3.14 (left)); to be specific, ALLSAT takes $\sim$ 1.3 seconds more than the SAT solver to verify the same property for the largest dataset (5K VNFFGs). On the other hand, ALLSAT solver consumes less CPU, while the memory consumption is almost the same for both solvers. On the other hand, though ALLSAT solvers are slower than SAT solvers, the required time by ALLSAT solver to identify multiple breaches (especially for a larger number of breaches) is less than an SAT solver as we described earlier in Figure 3.10 (right). Hence, we can conclude that NFVGuard+ is not solver dependent, and hence a user should choose the solver based on his/her requirements (e.g., find multiple breaches at a time or one by one).

Parallel Execution of the Properties. We can reduce the required time by verifying the properties in a parallel manner. Though different approaches are used to parallel verification [94], Sugar unfortunately, does not support parallelization. Hence, we adopt the search space splitting technique [94] which adopts a similar logic as in other parallel verification approaches. Specifically, in our technique, we split the audit data across multiple CSP instances that implement the same property rather than splitting the search space. In this way, we reduce the payload of verifying one large CSP instance by verifying less volume of audit data, and we can run the CSP instances in parallel. We choose the virtual resource isolation property because its the most resource-consuming property in case of detecting all non-compliance breaches (refer to Figure 3.11 (right)), and we also evaluate using the largest dataset with 100K VMs. As shown in Figure 3.12 (right), the time for the first round (two CSP instances) is reduced by 55% and the required time continues to decrease until we reach a reduction of 99%.

3.7.2 Experiments with Real Data

We apply NFVGuard+ to the real data collected from a real infrastructure hosted at one of the largest telecommunications vendors. The examined part of the infrastructure is composed of two racks, connected to two edge switches, which are connected to two aggregate switches, as depicted in Figure 3.15. The data contains 20 tenants, 111 VMS, 9 subnets, 26 physical servers, 26 vSwitches, 679 OvS flows, 35 VLANs, and 9 VXLANs. We apply NFVGuard+ to verify this real data against various properties. We report the average findings among those properties in Table 3.5. The resource consumption in terms of time, CPU, and memory increases with the amount of data as shown in Table 3.5. This result also follows a similar trend to what we found for the synthetic data in previous experiments. We can also observe that the values of resource consumption in this experiment are generally much smaller than in previous experiments performed using synthetic datasets (which were

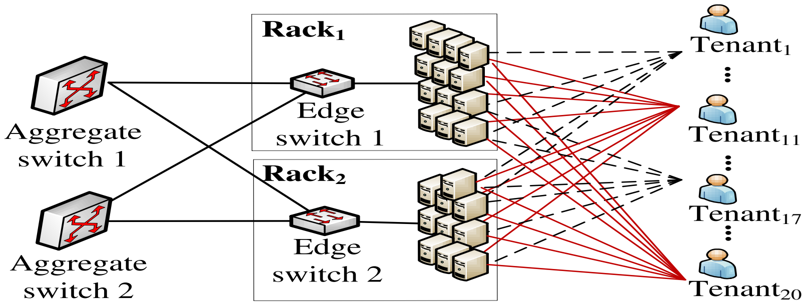


Figure 3.15: The topology of a part of a real cloud data center operating NFV used in our experiments.

deliberately scaled up to evaluate the scalability of our solution).

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

Table 3.5: The experimental results of NFVGuard+ for the real data. The average time, CPU, and memory required for the verification of three sample NFV security properties, i.e., VNFs co-residence, virtual resource isolation, and mapping unicity VLANs-VXLANs, based on real data.

3.8 Discussion

Complexity of NFVGuard+. The formal method employed in this work relies on solving Constraint Satisfaction Problems (CSPs), which are NP-complete. Consequently, in the worst case, solving a CSP for large NFV systems may require exponential time relative to the size of the input, including the number of entities, constraints, and relationships. Cross-level relationships within the NFV stack introduce a significant number of variables and constraints, further increasing the size and complexity of the CSP instance. Each level has its own policy constraints, and ensuring inter-layer consistency adds additional logical dependencies. As the number of records (e.g., configuration entries) and constraints grows, the verification time correspondingly increases. While formal methods provide accuracy and rigor, this approach becomes impractical for real-time or large-scale verification unless appropriate optimizations are applied.

Practicality and Robustness of NFVGuard+. Although the real dataset used in our experiments is modest in size and lacks cross-level violations, we conducted this experiment to demonstrate the practical applicability of our verification solution. Specifically, it shows the system's ability to operate effectively on real-world data, even in the absence of complex violations, and confirms the soundness of the formal verification logic under such conditions. Importantly, the results indicate that our approach produces no false positives—a critical property in real deployments, where false alarms can lead to wasted time and resources. Overall, the experiment validates that the solution behaves as expected in realistic settings and provides a reliable foundation for broader deployment or future evaluations on datasets that include actual violations.

The robustness of our solution stems from its ability to perform full-scale verification across all configurations and layers of the NFV stack. This comprehensive coverage strengthens confidence in the overall correctness and consistency of the system by thoroughly exploring all possible cross-layer dependencies and misconfigurations. In contrast, traditional approaches often overlook the multi-layer nature of the NFV stack and the associated threats and vulnerabilities. Additionally, the robustness is reinforced through the system's ability to propagate verification results across layers using consistency checks. When one layer is verified and its results align with expected mappings or dependencies, this information can be reused to minimize redundant checks in related layers—enhancing both efficiency and resilience to misconfigurations. Finally, the use of formal modeling ensures soundness and supports modularity, allowing the approach to be applied in large-scale NFV settings without losing correctness.

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

The first level in the ER model represents the entities created at the service orchestration level after processing the network service design specification (NSD) from the NFV user/provider. At this level, platforms such as ONAP [69], OSM [95], and Tacker are employed to enable the design, creation, orchestration, and auto-scaling of services on top of the resource management and virtual infrastructure layers. In our model, we depict high-level components related to the network service itself (not on the deployed platform), therefore, our model at this level is general and can be extended to different deployments. However, the scripts to collect and correlate data may vary with different deployments, especially when the data needs to be extracted from the configuration files specific to the deployed platform.

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

For instance, we consider a specific virtual infrastructure level implementation mainly relying on VLAN and VXLAN as well-established network virtualization technologies and OvS as a widely used virtual switch implementation. Other platforms may support different virtualization technologies such as Generic Routing Encapsulation (GRE) [96] or Generic Network Virtualization Encapsulation (GENEVE) [97]. In this case, the entities of the ER model at this level will change but not significantly (e.g., replacing the VXLAN entity with

GRE), and the properties may either remain applicable or need to be modified or skipped. As an example, in the case of small to medium clouds, where VLAN tags are sufficient to implement all L3 virtual networks on top of the physical network, the ER model will be simplified, and the security properties related to the mapping between VLAN and VXLAN become unnecessary.

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

Scope of the Security Verification. The security verification is conducted at a specific level(s) based on the definition of the verified security property. For example, the "Mapping unicity VLANs-VXLANs" security property (defined in [6]) can be verified only at L3 of the NFV stack because the data relevant to it exists at that level. Other properties, such as the "SFC ordering and sequencing as defined by the specification" security property (defined in [6]) require collecting data from various levels such as L1, L2, or L3. In this case, it depends on the auditor to define the specifications of the property. Therefore, the verification is property-dependent and sometimes can extend to all levels to ensure the correctness of the security property in the entire stack.

Automated Implementation. Since NFVGuard+ works with a static snapshot of the NFV environment, to maintain the security of the audited system, it needs to run periodically or on-demand when a change is made to the system. To that end, setting the period between verifications could be critical: a large interval between two verifications could lead to undetected security breaches and a small interval might incur prohibitive overhead. Hence we intend to improve the efficiency of our approach by exploring and adopting incremental [39] or proactive [41] techniques. Moreover, our current approach requires some manual effort and expertise in constructing the ER model, identifying the security properties, and formally encoding them. Although most of these efforts is done only once, we aim to automate those processes in our future work.

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

3.9 Summary

We presented NFVGuard+, a novel approach to the formal cross-level security verification of the NFV stack. Specifically, we proposed a system entity-relationship (ER) model that captures the detailed mappings and the relationships between the NFV resources across different levels in the NFV stack and devised a system that offers an assisted solution for NFV users to identify and verify the NFV properties by leveraging the ER mode. We implemented a real NFV testbed using OpenStack/Tacker, integrated our solution into the testbed, and evaluated our approach through experiments using synthetic data and real data provided by one of the largest telecommunications vendors. The results confirmed the efficiency and real-life applicability of our approach.

Chapter 4

Machine Learning Meets Formal Method for Faster Identification of Security Breaches in Network Functions Virtualization

4.1 Introduction

By decoupling network functions from proprietary hardware devices, Network Functions Virtualization (NFV) allows network services to be implemented as software modules running on top of generic hardware or virtual machines. This new paradigm allows service operators to more easily deploy a multi-tenant NFV environment on top of an existing cloud infrastructure, and it also allows NFV tenants to accelerate the provisioning and deployment of their services. Due to such benefits, the popularity of NFV is on the rise, e.g., in the context of 5G and beyond, NFV has become one of the main technology enablers for operators to scale their network capabilities on-demand at a lower cost by virtualizing dedicated physical devices on top of existing clouds [3].

The benefits of NFV may come at the cost of increased complexity. To support the management and orchestration of multiple network slices belonging to different tenants on top of the same cloud infrastructure [98], NFV relies on a mixture of virtualization technologies, e.g., a Virtual Network Function (VNF) such as virtual firewall seen at tenant-level may correspond to several virtual machines (VMs) connected through Software-Defined Networking (SDN) at the cloud infrastructure level [3]. Such increased complexity may also increase the chance of incorrect (e.g., lack of sufficient network isolation between different tenants' network slices [99]) or inconsistent (e.g,. a virtual firewall VNF specified at the tenant level may be bypassed at the underlying cloud infrastructure level [1]) configurations that could leave the services or infrastructure vulnerable to security threats. Therefore, the timely identification of such misconfigurations is important to ensure the security of NFV environments.

To that end, formal method-based security verification solutions (e.g., [100, 26, 4, 35, 6, 36, 29]) can provide rigorous proofs about the compliance or violation (with counterexamples) of the configurations w.r.t. given security properties. However, a key challenge is that the sheer scale of virtual environments can render formal security verification too costly. For instance, a state-of-the-art security verification tool requires around 12 minutes to check whether a guest VM can access any SDN controller with merely 5,000 reachability queries [4]. Such a delay can become much more significant under large NFV environments, resulting in a wide attack window during which the services or infrastructure are left vulnerable. Moreover, the inherent complexity of formal methods [101] can leave little room for further performance improvement, e.g., the aforementioned tool [4] is already heavily optimized (new combined filter-project operator and symbolic packet representation are added to the back-end verifier).

Motivating example. We further illustrate this issue through an example. The left side of Figure 4.1 shows the simplified view of a large NFV environment where two tenants, Alice and Bob, host their Virtual Network Functions (VNFs). Suppose our goal is to verify network isolation, i.e., whether any of Alice's VNFs can reach any of Bob's (except what is explicitly allowed). Even the verification of such a simple property (all-pair reachability) can become expensive as NFV tenants may own a large number of VNFs. To make things worse, NFV and its underlying cloud infrastructure typically employ distributed and fine-grained network access control mechanisms (e.g., per-VM security groups in OpenStack [102]). Consequently, verifying the reachability of two VNFs/VMs may require inspecting many rules and configuration data scattered among various data sources (e.g., routing and NAT rules in virtual routers along the route, host routes of the subnets, and firewall rules implementing tenant security properties [29]).

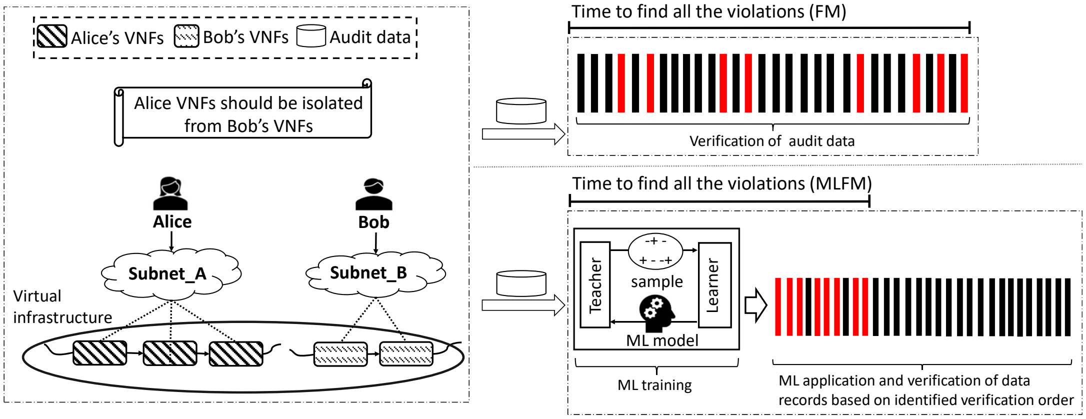

Alice's VNFs
Bob's VNFs
Audit data
Alice VNFs should be isolated from Bob's VNFs
Alice
Bob
Subnet_A
Subnet_B
Virtual infrastructure
Time to find all the violations (FM)
Verification of audit data
Time to find all the violations (MLFM)
Teacher
sample
Learner
ML model
ML training
ML application and verification of data records based on identified verification order

Figure 4.1: Motivating example

The right side of Figure 4.1 contrasts how the collected audit data will be processed under an existing formal method (FM)-based security verification approach (top) and under our approach (bottom). The barchart-like pattern illustrates the distribution of data records in the audit data where red (or black) bars represent pairs of VNFs that violate (or satisfy) the network isolation property. As the upper pattern shows, a FM-based approach would verify the audit data as is, i.e., all the VNF pairs will be verified in the same order as given in the audit data. In contrast, our approach leverages ML to reorder those data records such that those that (likely) cause violations (the red bars) will be moved forward, i.e., given a higher priority for verification than others (the black bars). Consequently, the verification can identify most of the violations in much less time (even after taking into account the time taken by ML training).

To that end, our main idea is to employ an iterative teacher-learner interaction, as depicted in the middle of Figure 4.1. In each iteration, the teacher (FM) first selects representative data records from the audit data, and then provides their verification results as training data to the learner (ML). Using such data, the learner (ML) trains an ML model, which is then given back to the teacher (FM) to be tested for identifying more representative data records (e.g., false positives and false negatives) in the next iteration. Over several iterations, such an interaction between the teacher and learner will enable a relatively accurate ML model to be trained using only a small portion of the audit data. The ML model can then be applied to reorder the remaining data for faster identification of violations.

4.2 Preliminaries

This section provides essential background on NFV, discusses NFV security properties, and defines our threat model.

NFV Background. NFV is a network architecture concept that decouples network functions (e.g., routers, firewalls, and load balancers) from proprietary hardware devices and virtualizes them as Virtual Network Functions (VNFs) running on top of existing cloud infrastructures [3]. Figure 4.2 presents a simplified view of the ETSI NFV reference architecture [3] (left), and an example NFV deployment corresponding to our motivating example (right). First, the *resource management* level conceptualizes the virtual resources such as subnets and VNFs. Second, the underlying *virtual infrastructure* level implements those virtual resources using virtual networking elements, such as virtual switches (e.g., *OVS_1*), VLANs (for communications within the same server), VxLANs (for communications between servers), and network ports, running on top of physical servers (e.g., *Server_1*). In this work, NFV configuration data stored in relational databases will be our main inputs.

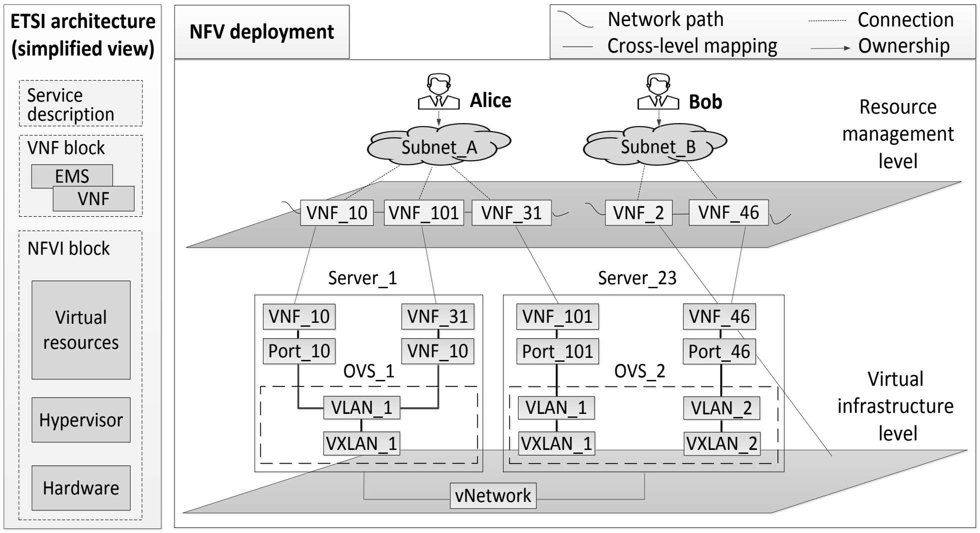


Figure 4.2: ETSI NFV reference architecture [3] (left) and an example NFV deployment corresponding to the motivating example (right)

NFV Security Properties. Various security properties can be defined to verify the compliance of NFV environments w.r.t. standards (e.g., ETSI [3] and IETF-RFC7498 [103]) or NFV tenants' requirements (Table 3.1 shows some example NFV security properties). Our approach can support other security properties as long as they can be verified using the chosen formal method tool (e.g., Sugar [67] used in this work can handle most properties formulated using standard first-order logic). To make our discussions more concrete, we describe two example properties (which will be needed later).

Example 4.2.1 First, the property *mapping unicity VLANs-VXLANs* ensures the logic segregation between different tenants' virtual networks through the unique assignment of VxLAN (communications between servers) identifier to each VLAN (communications within one server). Figure 4.3 (left) depicts a violation of this property (the shaded nodes show *VLAN_1* is mapped to both *VXLAN_10* and *VXLAN_16* on *Server_1*). Note this property can be verified for each VLAN separately. Second, the property *no VNFs co-residence* prevents a tenant's VNFs to be placed on the same physical server with VNFs of non-trusted tenants (e.g., due to concerns over potential side channel threats). Figure 4.3 (right) shows a violation of this property where *Alice's VNF_101* and *Bob's VNF_46* on both placed on server *S_23*. In contrast to the previous property, verifying this property could involve more records (all the VNFs of this tenant and the non-trusted tenants).

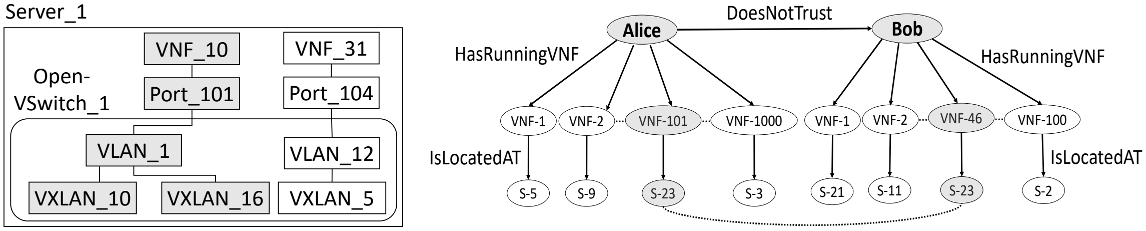


Figure 4.3: Two example NFV security properties: *Mapping unicity VLANs-VXLANs* (left) and *No VNFs co-residence* (right) (shaded nodes indicate violations)

Threat Model and Assumptions. Similar to most existing security verification approaches, our *scope* is limited to attacks that (directly or indirectly) cause violations to given security properties, and we assume our solution is deployed by the owner of the NFV environment who has access to the logs, databases, and configuration data needed for the security verification (and the integrity of those input data is protected with trusted computing techniques (e.g., [104])). Under such assumptions, our *in-scope threats* include both external attackers who exploit existing vulnerabilities in the NFV environment to violate the security properties, and insiders such as NFV operators and tenants who cause misconfigurations violating the properties, either through mistakes or by malicious intentions. Conversely, *out-of-scope threats* include attacks that do not cause any violation of the security properties, and attacks launched by adversaries who can erase evidences of their attacks by tampering with the logs, databases, etc.

We assume that the formal specification of security properties as well as the formal verification approach itself are correct and sound. As a security verification solution, our approach can only identify the violation of given security properties, but is not designed to attribute such a violation to the underlying vulnerabilities (responsibility of vulnerability analysis) or specific attacks (responsibility of intrusion detection). Similar to most existing machine learning approaches, we assume that a dataset required for verifying given security properties has been collected. However, we do not require labeled data, which can be difficult to obtain in a real world NFV environment, as the data records will be labeled by the teacher (formal method) in our approach (optionally, a small amount of labeled data records would be helpful for training an initial ML model to speed up the iterative approach). As with most security applications (e.g., spam or intrusion detection), we assume the dataset is unbalanced (i.e., the majority of data records belong to the compliance class w.r.t. the security property), and we make additional efforts in designing our approach to address this issue.

4.3 Methodology

This section first presents an overview of our approach, followed by details on the iterative teacher (FM)-learner (ML) interaction and the MLFM algorithm.

4.3.1 Overview

We propose a machine learning-guided formal security verification approach, namely, *MLFM*, for fast and provable identification of data records that violate a given security property in NFV. First, the *ML training* stage employs an iterative teacher (FM)-learner (ML) interaction to train an ML model using only a small portion of the audit data. Second, the *ML application* stage applies the ML model to reorder the remaining audit data, such that those that are more likely to violate the property will be verified first. More specifically, Figure 5.4 depicts our approach as follows.

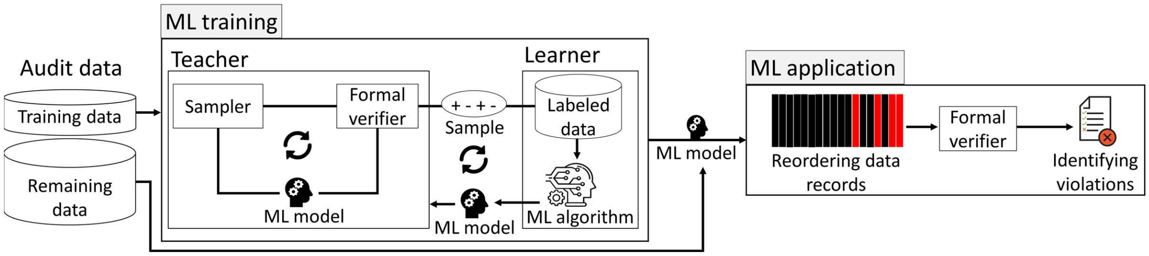


Figure 4.4: Overview of the MLFM approach

The ML Training Stage. As Figure 5.4 (left) shows, in each iteration of the teacher-learner interaction, the teacher first applies a sampling method to select a small data sample of fixed size from the audit data (shown as *Sampler* in the figure) after applying the ML model received from the learner in the previous iteration (an initial ML model is provided for the first iteration). The teacher then verifies the data records inside this data sample, and labels each record based on its verification result (shown as *Formal verifier* in the figure), and sends the labeled data sample to the learner. The learner then combines this newly received data sample with the previously received data samples to train a new ML model to be sent back to the teacher. This iterative interaction ends when reaching a predefined condition, e.g., a fixed iteration count, or lack of significant change in the accuracy of the model between two consecutive iterations.

The ML Application Stage. As Figure 5.4 (right) shows, the final ML model from the *ML training* stage is applied to the remaining audit data (i.e., the data not used for training) in order to identify data records that are more likely to violate the given security property, namely, the "to be verified" subset, which will be given a higher priority for verification. On the other hand, the "not to be verified" subset will either be verified afterwards, or not verified at all, depending on the use cases (detailed in Section 4.3.3).

4.3.2 Iterative Teacher (FM)-Learner (ML) Interaction

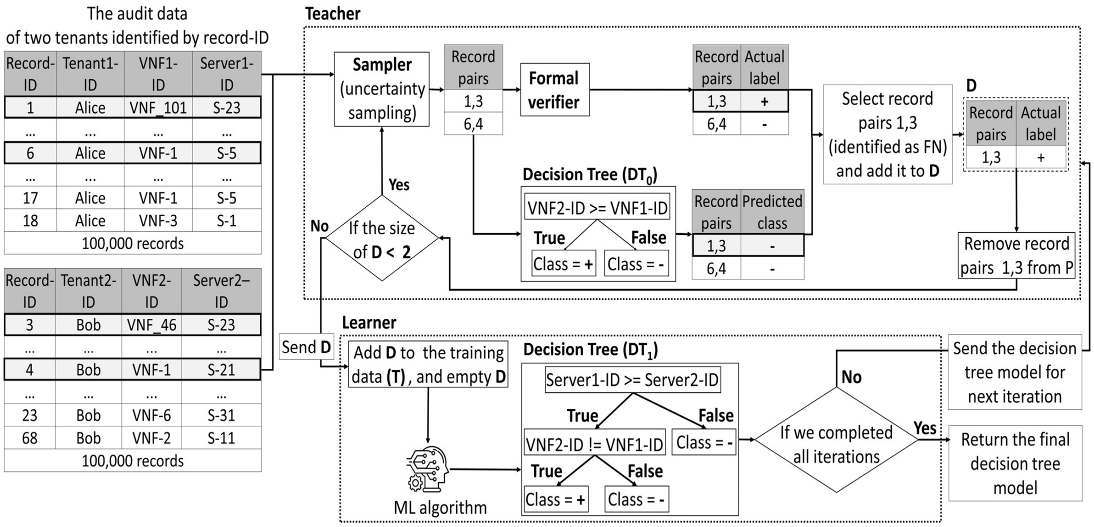
In the following, we provide more details about the key methodology of our approach, i.e., the iterative teacher (FM)-learner (ML) interaction.

Sampling (Teacher). The sampler component of the teacher is designed to select representative data records from the audit data in order for the learner to effectively enhance the ML model over each iteration. Choosing the right data records is important because they could cause either increase or decrease in the accuracy of the next ML model, e.g., data records having the same (redundant) information or those with the same label may cause the model to either not improve, or become biased towards the majority data, respectively. Our approach borrows sampling strategies (such as uncertainty sampling) from the active learning literature [46]. Although active learning has a different focus (it aims to reduce the effort of human experts in labeling the data, whereas no human expert is involved in our case), its sampling strategies are applicable to our approach, because they are also designed to better represent the characteristics of the property being analyzed such that an ML model can be trained with minimal labeled data.

Example 4.3.1 The left side of Figure 4.5 shows an excerpt of the audit data corresponding to the previous Example 4.2.1. Using uncertainty sampling, the sampler (inside the teacher block) selects a sample of size ($m = 2$) as the (shaded) record pairs $(1, 3)$ and $(6, 4)$.

Verification (Teacher). The formal verifier component is responsible for labeling the selected sample of data records (which will later be sent to the learner as training data). Labeling here means to annotate the data records with an extra field representing their classes, i.e., whether they are compliant with, or violate, the security property. To obtain such labels, the formal verifier performs formal verification by instantiating the security property (e.g., formulated using first-order logic) with the data records.

Example 4.3.2 Following Example 4.3.1, Figure 4.5 shows how the formal verifier labels



The audit data
of two tenants identified by record-ID

Record-ID	Tenant1-ID	VNF1-ID	Server1-ID
1	Alice	VNF_101	S-23
…	…	…	…
6	Alice	VNF-1	S-5
…	…	…	…
17	Alice	VNF-1	S-5
18	Alice	VNF-3	S-1
100,000 records			

Record-ID	Tenant2-ID	VNF2-ID	Server2–ID
3	Bob	VNF_46	S-23
…	…	…	…
4	Bob	VNF-1	S-21
…	…	…	…
23	Bob	VNF-6	S-31
68	Bob	VNF-2	S-11
100,000 records			

Record pairs
1,3
6,4

Record pairs	Actual label
1,3	+
6,4	-

Record pairs	Predicted class
1,3	-
6,4	-

D

Record pairs	Actual label
1,3	+

Teacher
Sampler (uncertainty sampling)
Formal verifier
Decision Tree (DT_0)
VNF2-ID >= VNF1-ID
True
False
Class = +
Class = -
Select record pairs 1,3 (identified as FN) and add it to D
Remove record pairs 1,3 from P
If the size of D < 2
Yes
No
Send D
Learner
Add D to the training data (T), and empty D
ML algorithm
Decision Tree (DT_1)
Server1-ID >= Server2-ID
True
False
VNF2-ID != VNF1-ID
Class = -
True
False
Class = +
Class = -
If we completed all iterations
No
Yes
Send the decision tree model for next iteration
Return the final decision tree model

Figure 4.5: An example of the iterative teacher (FM)-learner (ML) interaction

the selected sample by verifying the *No VNFs co-residence* property (see Section 5.6.1). Specifically, the formal verifier finds that the pair $(1, 3)$ violates the property (i.e., Alice's VNF (*VNF_101*) co-resides with Bob's VNF (*VNF_46*) on the same server (*S-23*)), and thus labels it as "+". The other pair $(6, 4)$ is labled as "-", as it does not violate the property.

Records Selection (Teacher). Next, the teacher applies the ML model from the previous iteration (received from the learner) to the labeled sample of data records. Intuitively, this allows the teacher to validate this previous ML model (by comparing its results to the labels provided by the formal verifier) and provide the "mistakes" (false positives and false negatives) as more representative training data to the learner. Specifically, as the ML model from the previous iteration also classifies the data records into two classes, by comparing its results to the ground truth, i.e., the labels assigned by the formal verifier component, the teacher can identify those records that have been correctly classified (i.e., true positives (TPs)) and those incorrectly classified (i.e., false negatives (FNs) and false positives (FPs)). Then, the teacher adds the TP, FN, and FP records to a new dataset D, which is the training dataset to be sent to the learner. Finally, if the number of records in D is still less than the desired size of the sample (m), the teacher repeats the aforementioned steps as an inner-iteration until it has accumulated totally m records in D. Note that the rationale for selecting (TP, FP, FN) records is twofold. First, as the positive class (i.e., violations) is generally smaller due to data imbalance, adding TP and FN records can augment the positive class to reduce the bias in training [105]. Second, the FN and FP records are incorrectly classified by the previous ML model and thus may contain more useful information for the learner to improve the accuracy of its next model.

Example 4.3.3 Following Example 4.3.2, Figure 4.5 shows a decision tree model (DT_0) received from the last iteration is applied to the two pairs of records $(1, 3)$ and $(6, 4)$. The decision tree (DT_0) predicts "+", if the *VNF2-ID* value is no smaller than the *VNF1-ID* value; otherwise, it is predicted as "-". Therefore, both $(1, 3)$ and $(6, 4)$ are predicted as "-". Comparing such results to the labels previously assigned by the formal verifier (see Example 4.3.2), we can see the pair $(1, 3)$ is *FN* and should be added to the dataset D (and deleted from the audit data), whereas $(6, 4)$ is *TN* and should not be added. Finally, as the size of *D* is less than the required size (*m*=2), we will repeat the inner-iteration.

ML Model Building (Learner). Once the teacher's dataset D reaches the required size m, the sample it contains is sent to the learner (D is then emptied in preparation for the next iteration). The learner adds the received sample to its existing training data (i.e., the collection of all previous samples), and utilizes this newly enriched training data to build a new ML model. The ML model is sent back to the teacher if the stopping condition (e.g., the specified number of interactions) has not been reached; otherwise, the interaction ends and the final ML model is given to the next (ML application) stage.

Example 4.3.4 Following Example 4.3.3, the lower part of Figure 4.5 shows that, once the teacher's inner-iteration ends, a sample of size two is sent to the learner. The learner adds the received sample to the existing training data (T) while the teacher empties its dataset (D). The new training data (T) is then used to build a new decision tree model (DT_1), which is more accurate than DT_0.

4.3.3 MLFM Algorithm and Use Cases

Algorithm 1 more formally states our approach. The inputs to the algorithm include the unlabeled audit data, the security property, and the parameters. The initial set of training data allows a system user to influence the algorithm with his/her domain knowledge by manually selecting/labeling data records (otherwise, the data can simply be randomly selected from the audit data and labeled using the formal verifier).

The algorithm has an outer iteration (Lines 2-9) which first builds a new sample through performing the inner iteration (Lines 3-7), and then adds this new sample to the existing training data (Line 8) to train a new ML model (Line 9). The outer iteration is repeated for a fixed number (provided as an input parameter) of times. The final ML model is then applied to reorder the remaining audit data before verifying it (Line 10). The union of all the verification results (Lines 5 and 10) is the final output.

The inner iteration builds a sample D of size m as follows. First, it selects a sample of size m from the audit data by following a given sampling strategy (Line 4). Although not shown in the algorithm, depending on the sampling strategy being used, this step may involve other parameters such as the current ML model (e.g., with uncertainty sampling [46]) or the training data (e.g., with Query-By-Committee (QBC) sampling [46]). Second, the sample is verified and labeled (with the verification results) using a formal verifier (Line 5). Third, the current ML model is applied to the sample, and the results are compared to the labels (verification results) to identify and add the (TP, FP, FN) records to D (Line 6). Fourth, D is removed from the audit data to avoid being selected again (Line 7). We repeat the above steps until D contains at least m records.

Complexity Analysis. The worst case complexity of the MLFM algorithm is $O(n \cdot (m \cdot (T_s + T_{v_1}) + T_t) + T_{v_2})$ where T_s, T_{v_1}, T_t, and T_{v_2} are the time for sampling (Line 4),

Algorithm 1: The MLFM algorithm

```
1  Inputs: Audit data (AD), security property (SP), initial training data ($T_0$), initial
   model $M_0$ = TrainClassifier($T_0$), per-iteration sample size (m), and iteration count
   (n)
   /* Outer-iteration                                          */
2  for $i$ = 0, $i < n$, $i$++ do
       /* Inner-iteration                                      */
3      while $| D | < m$ do
4          $S$ = SelectSample$(AD, m)$
5          $S_i$= VerifyAndLabel$(S, SP)$
6          $D = D \cup TP(S_i, M_i) \cup FP(S_i, M_i) \cup FN(S_i, M_i)$
7          $AD = AD \setminus D$
8      $T_{i+1} = T_i \cup D; D = \phi$
9      $M_{i+1}$ = TrainClassifier($T_{i+1}$)
10 return Verify(Reorder$(AD, M_n)$) $\cup$ ($\bigcup_i S_i$)
```

verifying m records (Line 5), training (Line 9), and verifying remaining records (Line 10), respectively. Such times would depend on specific algorithms, e.g., T_s under uncertainty sampling [46] can be estimated as $O(| \ AD \ |)$, since this strategy requires applying the current ML model on the audit data AD. T_{v1} and T_{v2} under a CSP solver is known to be exponential in the number of variables of the instantiated security property [106]. Finally, T_t under a decision tree classifier is $O(n_a \cdot n_t \cdot \log_2(n_t))$ [107] where n_a is the number of attributes and n_t the size of training data (i.e., $O(n{\cdot}m)$). We will further study the efficiency of the algorithm through experiments in Section 5.6.

Use Cases. Depending on how the remaining data is verified in Line 10 of the MLFM algorithm, our approach can be applied for two different use cases. First, MLFM running in the *partial verification* case will stop after verifying all the “to be verified” records (which would appear first after the reordering). This can be useful when the system user wants to find violations as quickly as possible (but not necessarily to find all the violations), and our objective in the training is to find an ML model that is the most accurate (since the misclassifed violations would not be verified, as further explained in Section 5.6). Second, MLFM in the *priority-based verification* case will verify all the records (with the "to be verified" records verified first). Our objective of the training is to find an ML model that incurs the least overall verification time with acceptable accuracy (since the mis-classified records will still be verified eventually).

4.4 Implementation

In this section, we describe the architecture and details of our implementation.

System Architecture. Our implementation of MLFM (shown in Figure 4.6) interacts with an OpenStack/Tacker [70]-based NFV environment to collect audit data. The system also interacts with a user to obtain other inputs, such as the security property to be verified, the formal verifier and the ML model to be applied, and the system parameters (the number of iterations and the sample size, as detailed in Section 4.3.3). Finally, the system returns an audit report to the user.

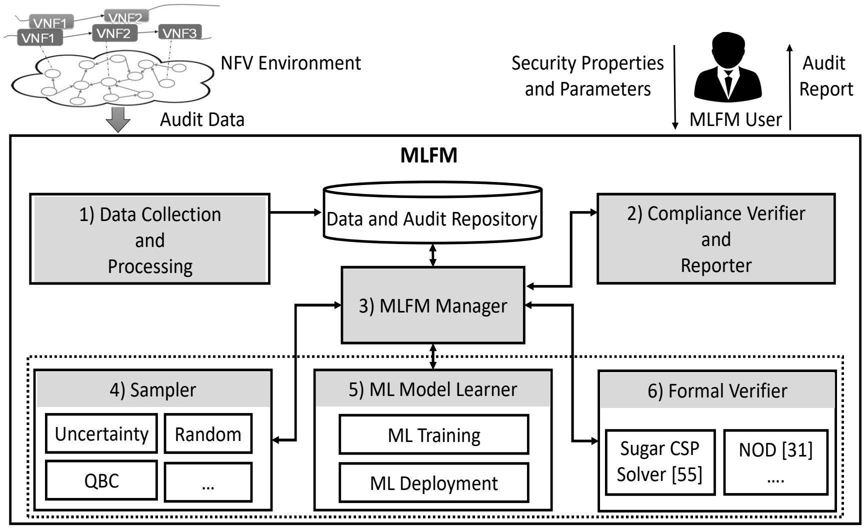


Figure 4.6: The MLFM system architecture

Data Collection and Processing. We implement this module using Python and Bash scripts to collect audit data from multiple sources including logs and configuration databases or files. For instance, to verify the *No VNFs co-residence* property, the module collects the identifiers of VNFs from Tacker and Nova databases [2], their corresponding owners (from Nova database), and the identifiers of servers hosting those VNFs (from Nova database). As the audit data are usually scattered among different components of the NFV environment and stored in different formats, the data must first be pre-processed. For instance, to verify the *mapping unicity VLANs-VXLANs* property, the data collected from OpenFlow tables of the OVS databases has unnecessary fields (e.g., *cookie* and *priority*) that must be filtered out. Also, the *port* and *vlan_vid* fields must be correlated to create the relation tuples *IsAssignedVLAN(ovs,port,valn)* for the verification. Finally, such filtered and correlated data must be converted into the corresponding input formats required by the formal verifier as well as for the ML training.

MLFM Manager. We implement this module in Python to manage and coordinate the interactions between other system modules for performing data collection and processing, data sampling, formal verification, ML training, etc., as described in Section 4.3.

ML Model Learner. We utilize Python 3.6.9 and Scikit-learn 0.24.1 (an open source ML library written in Python) to implement this module. We select decision tree, Support Vector Machine (SVM), and Random Forest (RF) models as they are among the most commonly used supervised classifiers, and are computationally more efficient compared to other classifiers such as K-Nearest-Neighbor (KNN) [108]. We also select XGBoost classifier [109], a scalable tree boosting system with a simpler structure using less resources than most other ML models, which has recently seen wide application for its high accuracy and low false positive rate [110, 111]. As our main aim is to reduce the overall delay before violations can be identified, we do not consider deep learning models as they are well known for higher complexity and longer training time compared to traditional ML models [112].

Sampler. We employ the *modAL* framework [113] to implement sampling strategies in this module. The *modAL* is an active learning framework for Python3, built on top of Scikit-learn [114], which allows to rapidly create active learning workflows with flexibility [113]. We select the uncertainty sampling and query-by-committee (with DT, SVM, and RF for members of the committee) sampling strategies in our implementation, as those are the most computationally efficient ones compared to other strategies [46].

Formal Verifier. We formalize the security properties together with the audit data as a Constraint Satisfaction Problem (CSP), a time-proven technique for expressing complex problems. Using CSP allows the user to specify a wide range of security properties (due to its expressiveness) in a relatively simple manner (as CSP enables to uniformly present the audit data as well as the security properties, and in a comprehensible and clean formalism, such as first order logic (FOL) [115]). Moreover, there exist many powerful and efficient CSP solver algorithms to avoid the state space traversal [116], which can make our approach more scalable for large NFV environments.

Once formulated as a CSP problem, the security verification is performed using Sugar [67], a well-established SAT-based constraint solver. We choose Sugar as it is an award-winning solver of global constraint categories (at the International CSP Solver Competitions in 2008 and 2009 [117]). Sugar solves a finite linear CSP by translating it into a SAT problem using order encoding method, and then solving the translated SAT problem using the MiniSat solver [118], which is an efficient CDCL SAT solver particularly effective in narrowing the search space [119]. Adapting our MLFM framework to other verification methods (such as theorem proving, model checkers, temporal logic, and Datalog) based on the needs of verification tasks is regarded as a future work.

Example 4.4.1 The predicate that corresponds to the negation of the *No VNFs co-residence* property is formulated (by the system user, done only once) as Formula 1 (left), and a predicate instance returned by Sugar to indicate violation is shown as Formula 2 (right) (i.e., both *Alice* and *Bob* have VNFs co-residing on the same server *S_23*).

$$\forall \texttt{t1}, \texttt{t2} \in \texttt{Tenant}, \forall \texttt{vnf1}, \texttt{vnf2} \in \texttt{VNF}, \forall \texttt{s1}, \texttt{s2} \in \texttt{Server} : \texttt{HasRunningVNF(t1, vnf1)} \land \texttt{HasRunningVNF(t2, vnf2)} \land \texttt{DoesNotTrust(t1, t2)} \land \texttt{IsRunningOn(vnf1, s1)} \land \texttt{IsRunningOn(vnf2, s2)} \land (\texttt{s1} == \texttt{s2}) \tag{1}$$

$$\texttt{HasRunningVNF(Alice, VNF\_101)} \land \texttt{HasRunningVNF(Bob, VNF\_46)} \land \texttt{DoesNotTrust(Alice, Bob)} \land \texttt{IsRunningOn(VNF\_101, S\_23)} \land \texttt{IsRunningOn(VNF\_46, S\_23)} \land (\texttt{S\_23} == \texttt{S\_23}) \tag{2}$$

4.5 EXPERIMENTS

This section describes the datasets and experimental settings, and presents our results.

4.5.1 Datasets and Experimental Settings

We first describe the implementation of our NFV testbed and data generation using the testbed, and then detail the experimental settings.

NFV Testbed Implementation. We choose to build our NFV testbed using OpenStack [2] with Tacker [70] mainly due to their growing popularity in real world [68] (other options such as Open Baton [120], OPNFV [121], and OSM [95] are still at their development stages). More specifically, we rely on the latest version OpenStack Rocky [2] for managing the virtual infrastructure, and we employ Tacker-0.10.0 [70], an official OpenStack project, to deploy virtual network services. Our NFV testbed consists of 20 tenants and 200 VNF forwarding graphs (VNFFGs), with each tenant owning around 10 VNFFGs and each VNFFG consisting of about 10 VNFs.

NFV Data Generation. To evaluate the performance of MLFM under large scale NFV environments, we would require a large scale NFV deployment. However, to the best of our knowledge, there do not exist any publicly available large-scale NFV deployment datasets. Therefore, we develop Python scripts to automatically generate various VNF Descriptors (VNFDs) and VNFFG Descriptors (VNFFGDs), which are then uploaded (also called onboarding) to our NFV testbed to deploy different network services and generate large scale

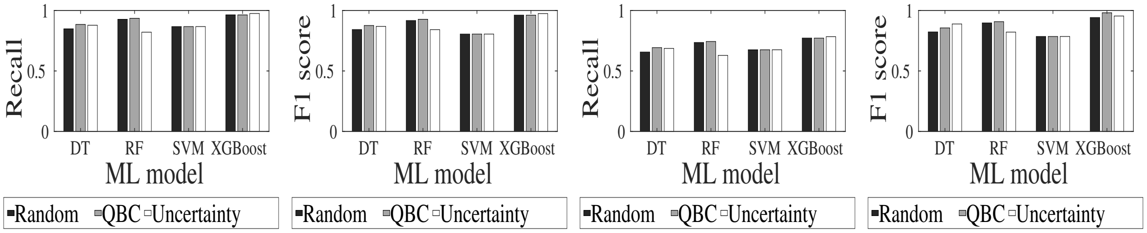
NFV datasets. We randomize parameters of those descriptors to ensure diversity in the generated data (e.g., the number of network ports per VNF, the flavor of each VNF, the number of VNFs in each Network Function Path (NFP), and the number of NFPs in each VNFFG). Our first dataset, *DS1*, contains 12,500 audit data records for verifying the *mapping unicity VLANs-VXLANs* property (P1 henceforth), and our second dataset, *DS2*, contains 25,000 records for verifying the *no VNFs co-residence* property (P2 henceforth). Each dataset contains around 10% of (uniformly distributed) records that violate the corresponding property.

Experimental Setting. All experiments are performed on a SuperServer 6029P-WTR running the Ubuntu 18.04 operating system equipped with Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz and 128GB of RAM without GPUs. All the experiments are performed using Sugar [67] as the formal verifier (unless mentioned otherwise) and Python 3.6.9 with Scikit-learn 0.24.1 ML packages for the ML method. For all the experiments, we use the default parameters for the ML models. Each experiment is repeated 1,000 times to obtain the average results.

4.5.2 Experimental Results

Best Performing Combination of ML Model/Sampling Method. The first set of the experiments aims to find the best performing combination of ML model and sampling method (as components of MLFM), from both the accuracy and time performance point of views. Specifically, Figure 4.7 shows the recall and F1 score results for different combinations of ML models (DT, RF, SVM and XGBoost, trained on 20% of each dataset) and sampling methods (random sampling, query-by-committee (QBC), and uncertainty sampling) for both security properties (P1 and P2) and datasets (DS1 and DS2). The results in Figures 4.7 (a) and (b) show that the combination of XGBoost and uncertainty sampling allows MLFM to achieve the highest recall (0.97) and F1 score (0.97) for security property P1. On the other hand, SVM combined with any of these sampling methods has the lowest F1 score

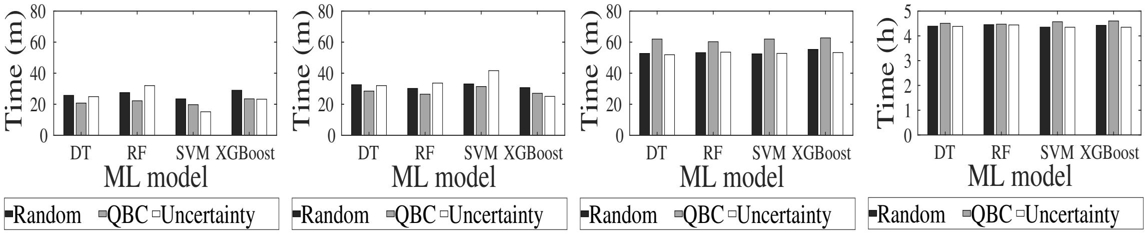
(0.80) (i.e., less effective in identifying both classes), and RF with uncertainty sampling has the lowest recall (0.82) (i.e., less effective in identifying the violations). Similarly, Figure 4.7 (c) shows that XGBoost with uncertainty sampling also has the best recall (0.783) for security property P2. However, as Figure 4.7 (d) shows, XGBoost has the best F1 score (0.981) when paired with QBC sampling. Nonetheless, as identifying the violations is more important to MLFM, XGBoost with uncertainty sampling is considered the best option for both P1 and P2.



(a) Recall for P1 (b) F1 score for P1 (c) Recall for P2 (d) F1 score for P2

Figure 4.7: Recall and F1 score for combinations of ML models and sampling methods, trained on 20% of dataset DS1 for property P1 (a and b) and on DS2 for P2 (c and d)

Figure 4.8 shows how the combinations of ML models and sampling methods affect the running time (in minutes) of MLFM (including both the ML training and application stages). As explained in Section 4.3.3, the partial verification use case aims to find the majority of violations in the least time. To that end, Figure 4.8 (a) seems to suggest that SVM paired with uncertainty sampling is the best option as it requires the least time (15.14 minutes). However, upon further investigation, this is not really the case, because the lower time consumption is mainly due to its inaccuracy (it misses more violations and thus, similar to most SAT solvers, Sugar incurs less time when there are less violations to find [67]). Therefore, considering both the accuracy (Figure 4.7 (a)) and the running time, XGBoost with uncertainty sampling seems to be the best option (with the second least time) for partial verification under P1. Figure 4.8 (b) shows that XGBoost with uncertainty sampling is the best option for priority-based verification for P1, as it requires the least time (accuracy is less important in this use case as all the records will be verified eventually,

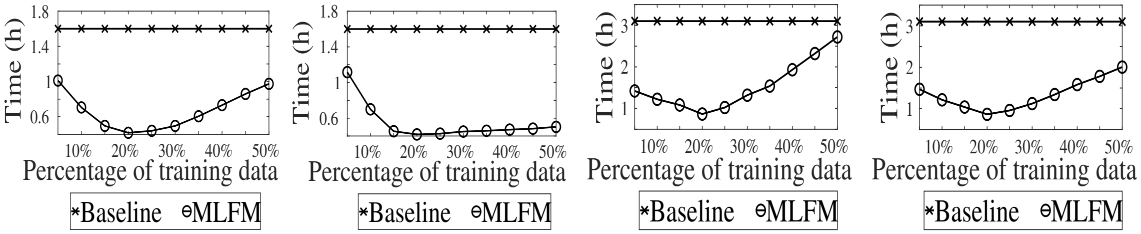


(a) Partial verification time for P1
(b) Priority-based verification time for P1
(c) Partial verification time for P2
(d) Priority-based verification time for P2

Figure 4.8: Running time of MLFM for combinations of ML models and sampling methods, with 20% of training data under P1 (a) (b), or P2 (c) (d), for both use cases

as explained in Section 4.3.3). Similarly, Figures 4.8 (c) and (d) show XGBoost with uncertainty is also the best combination under P2 for both use cases.

Best Performing Parameters m and n. In this set of experiments, we aim to find the optimal parameters of MLFM, i.e., the number of iterations n and the sample size m (see Section 4.3.3), in terms of the running time for priority-based verification, and also in comparison to the baseline approach (i.e., directly applying the formal verifier to the entire dataset). Specifically, Figure 4.9 (a) shows how changing the sample size m with a fixed number of iterations ($n = 10$) impacts the time, with the best performing model (i.e., XGBoost with uncertainty sampling) under property P1. The results show that MLFM takes less time (<1 hour) than the baseline approach (around 1.6 hours) in all cases. As more training data is used (through larger samples), the time of MLFM initially decreases due to more accurate ML models, and it reaches the lowest value (0.417 hr, or around 25% of the time of baseline) while using about 20% of the dataset for training. The time starts to increase afterwards, since the time needed to verify larger samples in the training stage becomes dominant (compared to the time saved in the application stage). Figure 4.9 (b) shows how changing the number of iterations n with a fixed sample size ($m = 250$) impacts the time. Similarly, MLFM takes less time than the baseline approach in all cases. The optimal percentage of training data is also around 20% (where $n = 10$). However, afterwards the time of MLFM stays lower than in the previous experiment, which shows that increasing the number of iterations is a safer choice (than increasing sample size) for increasing the training data. Figures 4.9 (c) and (d) show similar trends for property P2 (the longer time is due to more records involved in verification, as shown in Section 4.2).



(a) (b) (c) (d)

Figure 4.9: Running time of MLFM vs. the baseline (FM only) under property P1 (a) and (b) or P2 (c) and (d), using different percentages of training data either by changing the sample size m (a) and (c) or by changing the number of iterations n (b) and (d)

Comparing MLFM to Other Approaches. In this set of experiments, we compare the performance of MLFM to both the baseline approach (i.e., directly applying the formal verifier, Sugar [67]) and a state-of-the-art security verification tool, NOD [4] [1]. All experiments use the best performing model and parameters (i.e., XGBoost with uncertainty sampling, 20% training data, $m = 250$, and $n = 10$).

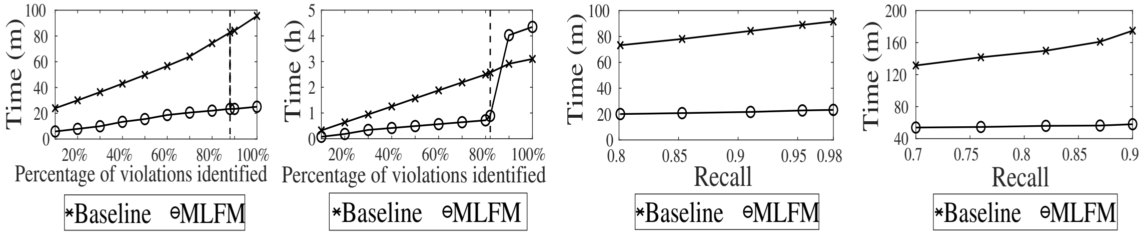
First, Figures 4.10 (a) and (b) show the time (in minutes) needed by the baseline approach (upper curve) and by MLFM (lower curve) for identifying different percentages of violations under properties P1 (a) and P2 (b), respectively. The figures depict both the priority-based verification use case (the entire curve) and the partial verification use case (part of the curve before the dashed line). Specifically, Figure 4.10 (a) shows that MLFM outperforms the baseline throughout the percentages, e.g., for partial verification, MLFM can identify 88% of the violations in around 23.3 minutes, which takes the baseline 82.7 minutes. Similarly, Figure 4.10 (b) shows that MLFM outperforms the baseline in case of partial verification for property P2, where it identifies 82% of the violations in about

[1] Among existing security verification tools, we do not compare to NFVGuard [6] as it actually forms the basis of our verification component, and we do not compare to TenantGuard [29] as it is based on custom algorithms instead of formal method.

53.3 minutes, while the baseline takes almost 2.4 hours. However, in case of priority-based verification (after the 82%) for property P2, MLFM takes more time than the baseline. The reason lies in the difference between the two properties. As explained in Section 4.2, unlike P1 (which can be verified for each VLAN independently), P2 may involve all the VNFs of a tenant, which means the remaining 18% of violations can only be identified using the baseline approach. Fortunately, there exists an alternative solution, i.e., we run MLFM and the baseline in parallel, and terminate MLFM as soon as the baseline finishes (as we already have all the results). As Figure 4.10 (b) shows, this would allow MLFM to identify around 86% of violations faster than the baseline, while bounding the overall running time by what is taken by the baseline.

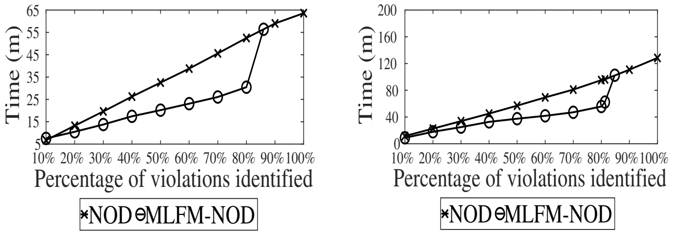
Next, Figures 4.10 (c) and (d) show the tradeoff between the running time (in minutes) and the recall values of partial verification (i.e., the percentage of violations identified by the end of partial verification) for P1 (c) and P2 (d). Both figures show similar results, i.e., while the baseline naturally requires more time for identifying more violations, MLFM can achieve a high recall value of 0.98 (P1) and 0.9 (P2) (by increasing the percentage of training data from 10% to 20%) with negligible change in running time (the difference will be greater for verifying the remaining records, as shown in Figure 4.9).

Finally, Figures 4.11 (a) and (b) show the time (in minutes) needed by NOD [4] (lower curve) and MLFM integrated with NOD (upper curve) for identifying different percentages of violations under the *virtual network reachability* property [4] (as this property is similar to P2, we run MLFM in parallel with NOD, as discussed above). We use the benchmarks provided in [4] to create two datasets with 25,000 and 50,000 reachability pairs, respectively, and around 10% of violations injected randomly. The results show that MLFM can help NOD to identify around 80% (a) and 81.3% (b) of violations, respectively, in less (57% and 65%, respectively) time.



(a) (b) (c) (d)

Figure 4.10: The time (in minutes) for identifying different percentages of violations by MLFM and the baseline for P1 (a) or P2 (b). The tradeoff between running time and recall values of MLFM and the baseline for partial verification of P1 (c) or P2 (d)



(a) (b)

Figure 4.11: The time (in minutes) for identifying different percentages of violations by NOD [4] and by MLFM integrated with NOD, using 25,000 (a) and 50,000 (b) records

4.6 Discussion

Model Training and Parameter Selection. As with any machine learning training task, the percentage of training data required is not fixed and largely depends on the specific security property being verified—particularly its complexity and the number of its attributes. In our evaluation, we demonstrated that the selected candidate properties used similar training data sizes; however, this may vary for other properties with different characteristics. It is important to note that the selection of training parameters—such as the number of iterations (n) and the sample size (m)—is performed only once for each property, after which these parameters can be reused in future verification. A key advantage of our approach is its focus on minimizing the training data size in order to reduce training time, which is a crucial factor in the overall verification process. This efficiency is achieved through the iterative teacher-learner approach, which ensures that even for varying properties, the model can be trained effectively with a small amount of data—making our approach practical, scalable, and time-efficient.

Significance of MLFM in Practice. NFV configurations are highly dynamic and often change at runtime; therefore, security verification needs to be fast, ideally completing within seconds to a few minutes, to detect or prevent issues early. MLFM supports near-real-time, on-demand security verification, making it suitable for compliance checks in large-scale NFV environments. It can be deployed continuously, aligned with configuration changes or deployment intervals, or periodically, such as every minute or few minutes, depending on the system's needs.

In practice, the MLFM approach is designed with scalability in mind, to overcome the scalability issue of formal verification and making it practical, efficient, and feasible for large-scale NFV deployments. Instead of replacing formal methods, MLFM enhances and guides them, enabling their utilization without alteration or compromising the accuracy of the security verification. Therefore, it improves the security without impacting system performance or introducing excessive overhead—an essential benefit for real-world NFV environments. Moreover, MLFM enables faster threat detection and reduces the verification backlog by prioritizing potentially high-risk configurations, allowing quicker responses to emerging vulnerabilities. These capabilities make continuous and real-time security auditing possible, which is critical for maintaining secure and resilient NFV systems.

Trade off Between Verification Accuracy and Efficiency. The trade-off between verification accuracy and efficiency is crucial in deploying security verification solutions—especially in real-world, large-scale environments such as NFV and cloud. achieving guaranteed accuracy with formal methods often comes at the cost of speed and scalability. MLFM addresses this by using machine learning to prioritize configurations that are more likely to violate security policies, allowing formal verification to be applied selectively and strategically. This significantly reduces verification time—from hours to minutes—and lowers resource usage by focusing on high-risk configurations. While it is less rigorous than exhaustive formal methods, it achieves acceptable accuracy, making it a practical solution for dynamic NFV and cloud environments. This is especially valuable in time-sensitive situations where verifying all the configuration is infeasible.

MLFM is also effective when auditors aim to reduce computational load, lack sufficient resources, or require rapid feedback for faster vulnerability response (i.e., partial verification use cases). Comprehensive formal verification can then be reserved for critical configurations or scheduled periodically during off-peak hours, since performing exhaustive checks on every update is often impractical in large-scale NFV deployments.

Furthermore, when 100% accuracy is required, MLFM offers a practical trade-off between efficiency and accuracy by prioritizing the verification of high-risk configurations first, followed by the verifying the remaining records (i.e., the priority-based use case). This approach ensures that critical threats are addressed promptly, reducing the window of vulnerability. Although some records may be misclassified and verified later, the entire dataset is eventually checked. As a result, verification time is significantly reduced without compromising overall security. In summary, MLFM provides a scalable, prioritized, and accurate verification strategy that aligns well with real NFV deployments, offering a balanced and highly practical solution.

4.7 Summary

We have presented MLFM, a novel approach to security verification in NFV that could combine the rigor of formal methods with the efficiency of machine learning for faster identification of security violations. Specifically, we designed an iterative approach for the teacher (FM) to gradually provide more representative data samples, such that the learner

(ML) could train an ML model using a small portion of the data; the ML model was then applied to the remaining data to prioritize the verification of likely violations. We implemented MLFM based on OpenStack/Tacker, and our experimental results showed significant performance improvements over baseline approaches.

Chapter 5

Security Verification for Microservices Using *F*ederated *L*earning-Guided *F*ormal *M*ethod

5.1 Introduction

The microservices architecture [122], a widely adopted software development architectural style, organizes and implements modern cloud applications as a collection of small, loosely coupled services—namely, *microservices*—with well-defined interfaces and operations. Each microservice can be developed, deployed, upgraded, and scaled independently, thereby significantly improving the scalability and flexibility of application development and maintenance at lower costs. As a result, the microservices architecture has become a de facto standard for developing large-scale commercial cloud applications, as evidenced by its adoption by many well-known companies (e.g., Uber [123], Twitter [124], and Netflix [125]).

Despite their benefits, microservices can pose novel security and privacy challenges.

Since real-world microservices-based applications (mApps) can become very large and complex (e.g., 1,000 microservices for the case of Uber [123], and $\mathcal{O}(10^5)$ different microservices scaled to $\mathcal{O}(10^3)$ service instances[1] for Twitter in 2016 [124]), they may suffer from a larger attack surface and a higher risk of misconfiguration compared to their monolithic counterparts. For instance, communications among microservices, which were previously conducted through local invocations within a monolithic application, are now exposed through the network, which increases the security risks for the entire application. Consequently, an adversary can exploit the exposed inter-service communications to attack the entire application by sending malicious requests from one compromised microservice to the other microservices [52]. The timely identification of such security threats is important to ensure the security of the microservices-based applications.

In this regard, formal method-based security verification solutions (e.g., [47, 100, 26, 4, 6, 5, 48, 49]) are widely adopted to provide rigorous evidence about the compliance or violation (alongside counterexamples showing security breaches) of configurations w.r.t. given security properties. However, applying formal methods to microservice applications may face two major challenges. First, the sheer scale of microservice applications can exacerbate the inherent complexity of formal methods [101] and cause them to experience significant delay in identifying security breaches. This delay could result in a wide attack window inside which the microservice applications will remain vulnerable. Second, due to the distributed nature of microservice applications, it may be infeasible to perform security verification at a central location. Collecting configuration data from all the microservices could either be expensive (in terms of high communication costs) or impossible (e.g., due to data confidentiality and privacy concerns, considering the fact that such configuration data may be highly sensitive as they may reveal security flaws, and are typically governed by different organizations or administrative domains who are reluctant to share such data).

[1] In production, administrators may use multiple identical instances of a microservice to improve performance and provide high availability [52].

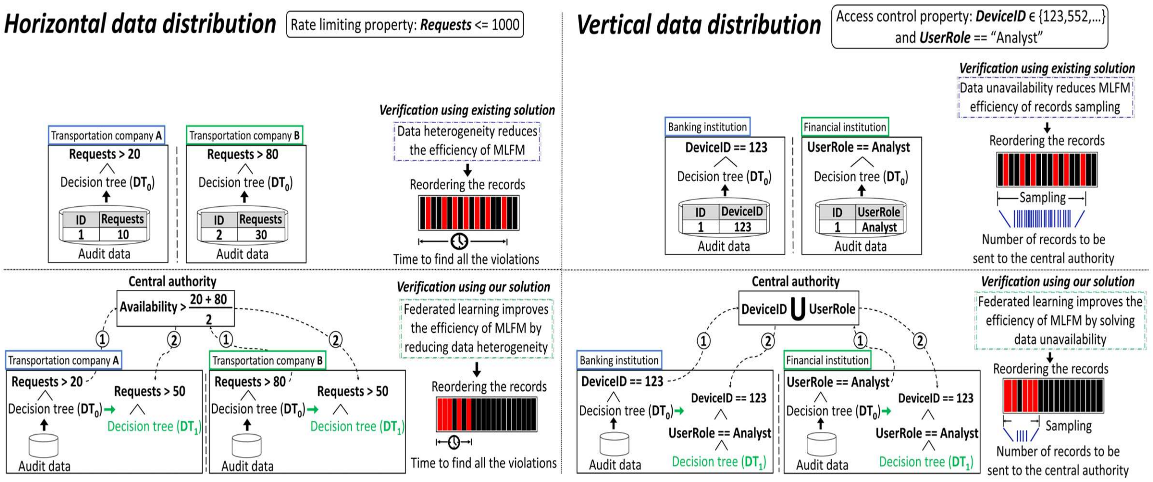


Figure 5.1: Motivating example

Motivating example. We further illustrate this issue through an example. Specifically, Figure 4.1 shows two cases of data distribution, where the *horizontal* case (left) means the audit data of the two microservice applications share the same attributes (i.e., *ID* and *Requests*) but may have different ranges of values (e.g., assume *Transportation company A* tends to have higher requests rate than *Transportation company B*), and the *vertical* case (right) means the two applications have different attributes (i.e., *ID* and *DeviceID* for *Banking institution*, but *ID* and *UserRole* for *Equifax*). The top of each side shows the property to be verified, while the middle and bottom demonstrate the challenges faced by an existing solution [5], and the key ideas of our solution, respectively.

More specifically, for the horizontal case (left), suppose we need to verify whether the *Transportation company A* and *Transportation company B* microservice applications are compliant with the following property, i.e., the number of `Requests` per second should be less than or equal to 1,000. To that end, MLFM [5] is an existing work designed to reduce the verification delay through guiding the formal method verification using a machine learning model. However, as illustrated in the figure (middle left), since the *Requests* value of the *Transportation company B* tends to be higher than that of the *Transportation* *company A* application, the machine learning model trained locally using each individual application's data will not be very accurate due to such data heterogeneity. Consequently, the model is ineffective in prioritizing (through reordering) the data records such that likely security breaches (represented by red bars in the figure) can be verified first.

In contrast, our solution (lower left) leverages Federated Learning (FL) such that the two applications can transmit local model parameters (instead of the raw data) to the central authority, which then aggregates such parameters to obtain a more accurate FL model, e.g., by taking the average of the *Requests* thresholds (Step 1). This FL model can then be applied by each application to more effectively prioritize (reorder) its data records to reduce the verification time. The vertical case (right) is similar, except that the challenge faced by MLFM [5] is caused by data unavailability (as each application lacks certain attributes) instead of data heterogeneity, and that the central authority needs to perform global verification (in addition to parameter aggregation), which will be detailed later in Section 5.4.

5.2 Preliminaries

This section provides necessary background information and defines our threat model.

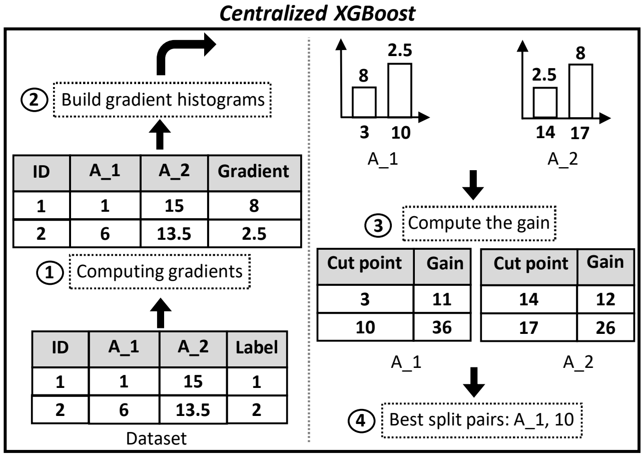
5.2.1 XGBoost

In this work, we employ XGBoost as the main ML model (although our methodology can be extended to other models). XGBoost [109] is an optimized implementation of the Gradient Boosted Decision Trees (GBDT) which utilize an ensemble of sequentially trained decision trees to make the predictions. The training of a single tree would require to first calculate the gradient (g) and hessian (h) values for each data sample. Then, at the node level, it builds a gradient (and hessian) histogram for each attribute and finds the value that maximizes the gain i.e., the value that best split the samples by maximizing the loss reduction after split. Next, it finds the attribute with the maximal gain for the current node and splits the samples accordingly. If the tree reaches a predefined restrictions such as the maximum depth or the gain being always smaller than zero, then the current node is considered as a leaf node that holds the prediction result. The gradient (and hessian) histogram show a small number of cut points as possible split candidates. The samples are sorted based on their attribute values and arranged into **q** buckets corresponding to the cut points. The gradient (or hessian) of each cut point is the sum of the gradients of the samples that fall into its bucket.

Figure 5.2 shows an example of finding the best split for a single node. Given the dataset identified by record *ID*, the gradients (and hessians, which are omitted for simplicity) are first calculated. Then, the gradient histograms are built using the calculated gradients. The histogram of *A_1* has two cut points (3,10), the gradient of each cut point consists of the sum of the gradients of the samples that fall into its bucket e.g., the sample identified by *ID* (1) falls under that cut point (3). Since the value of its attribute (i.e., *A_1*) is less than the cut point (3). Therefore, the gradient of cut point (3) is the gradient of the sample, i.e., (8). Afterward, the gain is calculated for each cut point of the histograms, and the cut point with the highest gain for each histogram is chosen i.e., (10) for histogram of *A_1* and (17) for histogram of *A_2*. Finally, the attribute with the maximal gain is chosen i.e., *A_1*. Accordingly, the current attribute-value pairs that best split the dataset for the current node are (*A_1*,10).

5.2.2 Federated Learning

Federated Learning (FL) enables collaborative decentralized privacy-preserving training of machine learning models among multiple parties to address the data privacy and security issues [126] raised by regulations (e.g., the General Data Protection Regulation



ID	A_1	A_2	Gradient
1	1	15	8
2	6	13.5	2.5

ID	A_1	A_2	Label
1	1	15	1
2	6	13.5	2

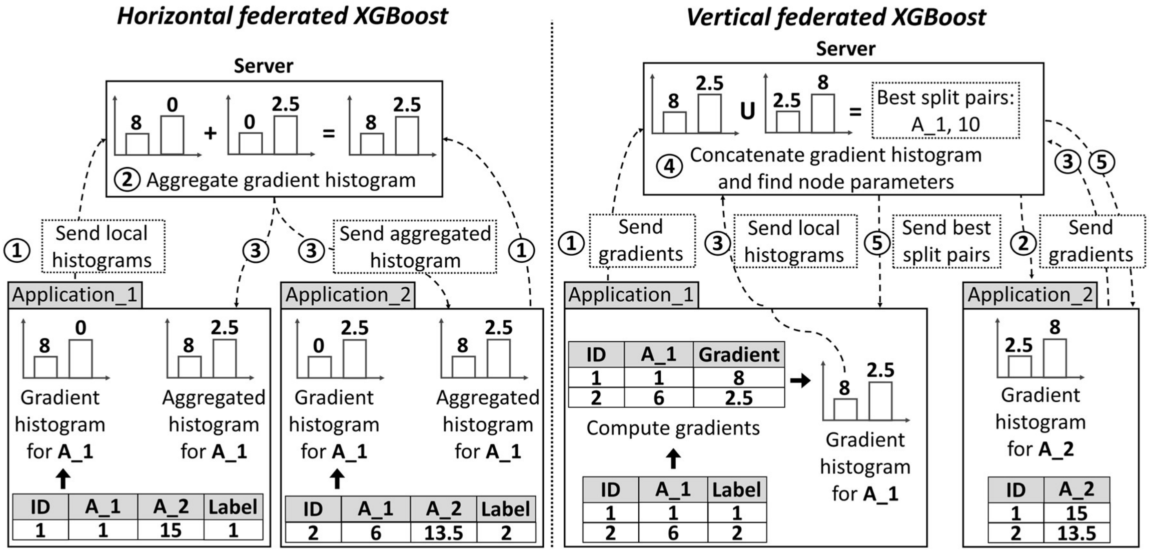
Cut point	Gain
3	11
10	36

Cut point	Gain
14	12
17	26

Figure 5.2: An example illustrating XGBoost training

(GDPR) [127], Health Insurance Portability and Accountability Act (HIPAA) [128], and California Consumer Privacy Act (CCPA) [129]). FL generally requires the participants to train a local ML model with their local data, and send the local model parameters (also called local updates) to a central server through a secure channel. The server (also known as aggregator or coordinator) fuses the local updates to create a global model, which is often sent back to the participants. This process could be repeated for several iterations to achieve a desired performance, or until reaching a stopping condition. Federated learning can be categorized into Horizontal Federated Learning (HFL), Vertical Federated Learning (VFL), and Federated Transfer Learning (FTL) according to the distribution of data [130]. In this work, we focus on the HFL and VFL cases. Specifically, HFL means all participants share the same attributes, but they may have different data samples. In contrast, VFL means all participants share the same data samples, although they may have different attributes.

Figure 5.3 (left) shows an example of XGBoost training in the HFL settings (most of the solutions that utilize XGBoost in the FL settings exchange the gradients, hessians, and/or attribute-value split candidates between the parties and the server [131, 132, 133], and we focus on the gradients and hessians histograms in this work). Specifically, the dataset (bottom left) is divided horizontally between *Application_1* and *Application_2*. First, in each


Horizontal federated XGBoost
Server
8
0
+
0
2.5
=
8
2.5
② Aggregate gradient histogram
①
Send local histograms
③
③
Send aggregated histogram
①
Application_1
8
0
Gradient histogram for A_1
8
2.5
Aggregated histogram for A_1
Application_2
0
2.5
Gradient histogram for A_1
8
2.5
Aggregated histogram for A_1
Vertical federated XGBoost
Server
8
2.5
U
2.5
8
=
Best split pairs: A_1, 10
④ Concatenate gradient histogram and find node parameters
①
Send gradients
③
Send local histograms
⑤
Send best split pairs
②
Send gradients
③ ⑤
Application_1
Compute gradients
8
2.5
Gradient histogram for A_1
Application_2
2.5
8
Gradient histogram for A_2

ID	A_1	A_2	Label
1	1	15	1

ID	A_1	A_2	Label
2	6	13.5	2

ID	A_1	Gradient
1	1	8
2	6	2.5

ID	A_1	Label
1	1	1
2	6	2

ID	A_2
1	15
2	13.5

Figure 5.3: An example illustrating XGBoost training in the HFL (left) and VFL (right) settings

round, communicating for each node, the applications locally compute the histograms and then send the histograms to the server (Step 1). Then, the server aggregates the histograms through summation, i.e., the gradient (and hessian) values that fall inside a specific bin interval are summed within their respective value buckets (Step 2). Next, the applications receive the aggregated histograms from the server to update the local models (Step 3). Eventually, each application will end up with histograms that are created as if it had the entire dataset (e.g., the histogram of attribute *A_1* at the top of the figure is identical to the histogram of the same attribute on the left).

Figure 5.3 (right) shows an example of the XGBoost training algorithm in the VFL settings. The dataset (bottom right) is divided vertically between the applications. Only the application that has the labels (usually called the active participant), i.e., *Application_1*, can compute the gradients (and hessians). Therefore, this application sends the gradient (and hessian) values to the server (Step 1), once for each tree [134]. Then, the server sends these to the other applications that do not have the labels (*Application_2*) such that they can compute the corresponding gradients (and hessians) histograms (Step 2). Similar to the HFL case, in each round, communicating for each node, the applications send the histograms to the server (Step 3), which then aggregates the histograms through concatenation i.e., grouping together the histograms of all attributes, to compute the node parameters (Step 4). The best split (i.e., attribute-value pairs) or the prediction value if the current node is a leaf node. Finally, the applications receive the node parameters from the server and update the local models (Step 5). Each party will end up with the same global ML model (e.g., the split value at the top is the same as the one on the left).

5.2.3 Threat Model and Assumptions

Following the majority of federated learning and microservice approaches, we make the following assumptions. First, the microservice applications are properly isolated and can only communicate with the central server through a secure channel. Second, the data is independently and identically distributed (IID) among all the applications. Third, while FL protects the privacy of the applications by not exchanging the raw data, potential information leaks or inference attacks on the exchanged information (e.g., histograms) may still be possible and are assumed to be prevented using techniques such as homomorphic encryption or differential privacy. Also, following the security verification literature, we assume our solution is deployed by the microservice provider or owner, who has reliable access to the required logs, databases, and configuration data. We also assume that the formal specification of security properties, as well as the formal verification approach itself are both sound. Finally, as a security verification solution, our approach is only designed to identify security violations, and we assume it can work in tandem with other security solutions to attribute such violations to their root cause (e.g., vulnerabilities or attacks).

Under such assumptions, our *in-scope threats* include both external attackers who exploit existing vulnerabilities in the microservice architecture to violate security properties, and insiders such as cloud operators and tenants who cause misconfigurations that violate those properties, as well as the provider or owner of a microservice application who may be curious to learn about other applications' data. On the contrary, our *out-of-scope threats* encompass attacks that do not result in breaches of any security properties, attacks launched by adversaries capable of eliminating evidence of their actions (e.g., logs and databases), or tampering with the cloud infrastructure or compromising our solution itself, and attacks launched by malicious insiders such as cloud operators/tenants and microservice application providers/owners (e.g., deviating from the FL procedure, poisoning the training data with adversarial samples, or eavesdropping on the communication channel).

5.3 Horizontal FLFM (H-FLFM) Methodology

This section first presents an overview of our horizontal FLFM (H-FLFM) approach, and then details its components.

5.3.1 Overview

The horizontal case of FLFM (H-FLFM) is designed for verifying a security property across several microservice applications that share the same attributes but have different data values, e.g., one application tends to have larger values than other applications for the same attribute. In such a case, as illustrated in our motivating example (Section 5.1), the machine learning models of MLFM trained locally using each application's own data may not be accurate enough due to data heterogeneity. Therefore, H-FLFM utilizes Horizontal Federated Learning (HFL) to enable the applications to collaboratively train a global ML model to reflect more accurately the combined data of all applications. As detailed in

Section 5.2.2), HFL allows the global model to be trained without the need for sharing each application's data with either the central authority or between the applications themselves. Instead, only certain learning parameters will be exchanged between the applications and the central authority, thereby preserving the confidentiality and privacy for the applications.

At a high level, as demonstrated in Figure 5.4, H-FLFM consists of two stages. First, during the *ML training stage*, an iterative teacher (FM)-learner (ML) interaction (as detailed in [5]) is utilized to train a local ML model using a small portion of the applications' audit data. Subsequently, the learning parameters of each application are sent to, and aggregated by, the central authority. The aggregated parameters are then sent back to the applications to refine their models for higher accuracy. These steps are repeated until a predefined stopping condition is met. Second, in the *ML application stage*, the final ML model of each application is applied to the remaining local audit data (i.e., the data not used during training) to prioritize the verification of data records that are more likely to violate the property. The remainder of this section provides further details and examples of our H-FLFM methodology.

5.3.2 Training Stage - Local Model Training

The local ML model training occurs concurrently on each microservice application following the iterative teacher (FM)-learner (ML) interaction introduced in [5]. The goal is for the teacher to select a minimal yet representative (with respect to the current ML model) set of records in each iteration, such that the learner can obtain a relatively accurate ML model using a small amount of data. Specifically, as shown in Figure 5.4 (lower-right inside *Application_1*), in each teacher-learner iteration, the teacher employs a sampling method to select a small fixed-size data sample from the audit data (depicted as *Sampler*) by applying the ML model received from the learner in the previous round (an initial ML model is provided for the first iteration). This sample is verified by the *Formal verifier*

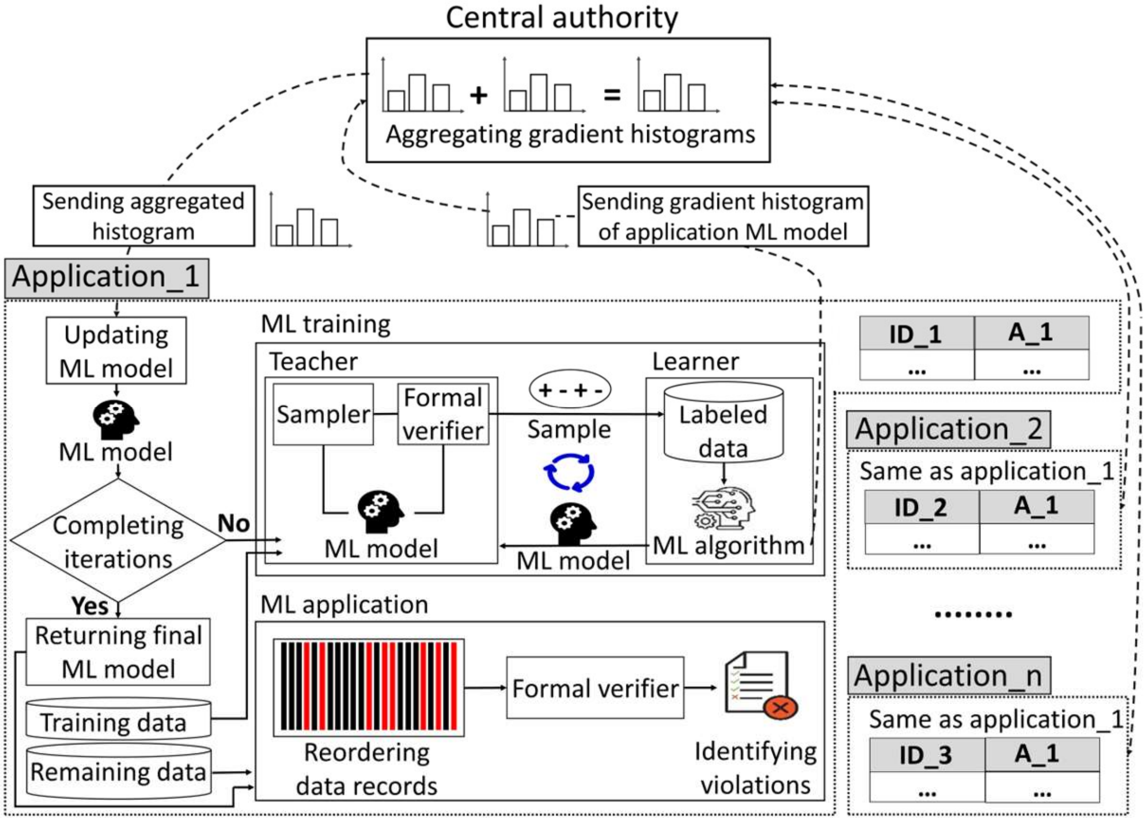


Figure 5.4: Overview of the horizontal FLFM approach

to label its records according to the verification results (depicted as *Labeled data*) before being sent to the learner. The learner adds this new sample of labeled data to the previously received ones to train a new local ML model, which is then sent back to the teacher for the next iteration. This iterative interaction ends when reaching a predefined condition (e.g., a fixed iteration count, or lack of significant improvement in the accuracy of the model).

Example 5.3.1 Figure 5.5 (Step ①) shows an example of the local ML training at both the *Uber* and *Bolt* applications. For *Uber* (lower middle), using the uncertainty sampling method, the sampler (inside the teacher block) selects a sample of size ($m = 1$), and assume data record (1) is selected. Then, the formal verifier verifies this record, and assume it assigns the $(+)$ label to the record. On the other hand, assume the local DT model classifies this same record as $(-)$. Therefore, record (1) is a FN record containing representative data, which is selected by the teacher to be part of D (which contains records to be sent to the learner and removed from the audit data P). Since we assume a required sample size of ($m = 1$) in this example, the sample is sent to the learner and added to the existing training data (T), while the teacher empties its (D). The learner uses (T) to build a new (more accurate) local ML model, which is sent back to the teacher for the next iteration, until the iteration count has reached a predefined count (n). The same process occurs in parallel at the *Bolt* application (only the differences are highlighted in the figure).

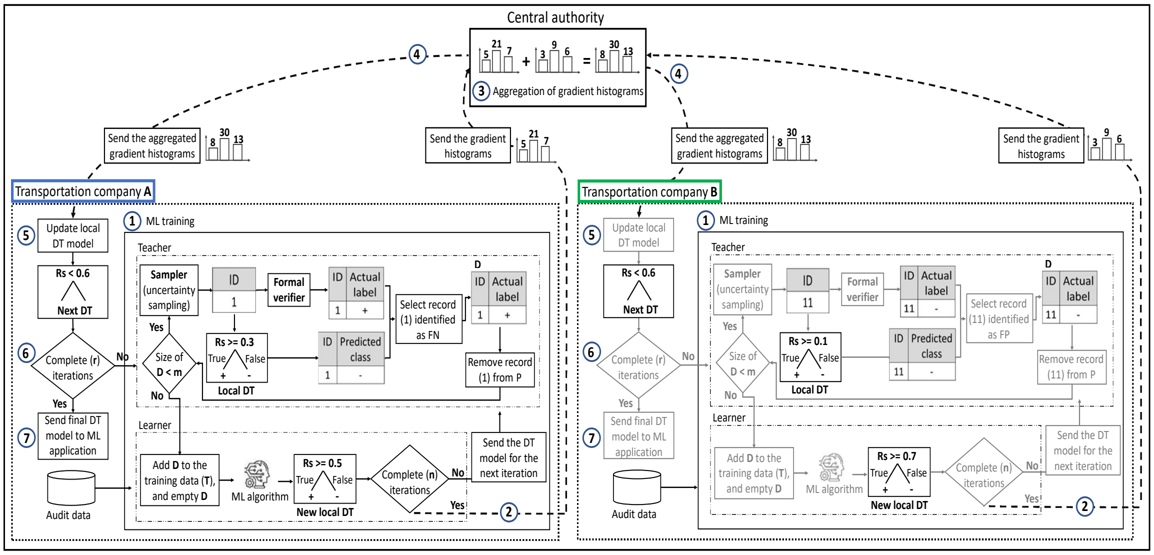


Figure 5.5: An example of the horizontal FLFM training

5.3.3 Training Stage - Global Model Learning

The federated global model learning starts at the end of each iteration of the teacher (FM)-learner (ML) interaction. The goal is for the applications to send their ML model parameters to the central authority, which then aggregates these parameters and sends the result back to the applications to refine their local ML models. Specifically, Figure 5.4 (lower left inside *Application_1* and top) illustrates these steps using a federated XGBoost classifier (detailed in Section 5.2.1). First, the applications send their learning parameters to the central authority, i.e., the gradients and hessians histograms (as detailed in Section

5.2.2). Second, the central authority performs an aggregation operation (i.e., summing the values within their respective value buckets in this case) on the gradient and hessian histograms received from all the applications to obtain a final histogram representation. The central authority then sends the final histogram back to each application. Next, each application uses the received histograms to update its local ML model to obtain a more accurate model that better reflects global knowledge. Finally, the updated local ML model is utilized by the next iterative teacher (FM)-learner (ML) interaction to locally train a new ML model. The central authority and the applications may repeat this process for several iterations until a stopping condition is reached (e.g., a predefined iteration count or when the histograms see no significant change between two consecutive iterations).

Example 5.3.2 Following Example 5.3.1, Figure 5.5 (Step ②) shows that, at the end of the n^{th} iteration of the teacher-learner interaction, each application sends its local ML model parameters (i.e., gradient and hessian histograms) to the central authority (for clarity, the figure only displays the gradient histograms). Specifically, two gradient histograms, each with three gradient values i.e., $(5, 21, 7)$ and $(3, 9, 6)$, are sent by the *Uber* and *Bolt* applications, respectively. Step ③ shows the aggregation of the received gradient histograms at the central authority. Particularly, the aggregation (i.e., summation) results are represented in a histogram with the values $(8, 30, 13)$, which are then sent back to the applications (Step ④). Step ⑤ shows that each application uses the received aggregated gradient histograms to update its local model. If a predefined number (r) of aggregation iterations is not yet reached (Step ⑥), the updated local model is provided to the teacher to trigger a new iteration of the teacher (FM)-learner (ML) interaction for local ML model training. Otherwise, the updated local model will be utilized by the ML application stage (Step ⑦).

5.3.4 Application Stage

As Figure 5.4 (bottom of *Application_1*) shows, the final ML model obtained from the *ML training stage* is applied to the remaining audit data (i.e., the data not used for the training) of each application in order to identify data records that are more likely to violate the given security property, namely, the "to be verified" subset, which is then given a higher priority for verification. On the other hand, the other records, i.e., the "not to be verified" subset, will either be verified afterwards, or not verified at all, depending on the use cases (discussed in Section 5.6).

5.4 Vertical FLFM (V-FLFM) Methodology

This section first presents an overview of our vertical FLFM (V-FLFM) approach and then details its components.

5.4.1 Overview

The vertical case of FLFM (V-FLFM) is designed for verifying a security property across several microservice applications that share the same data samples but with different attributes. In such a case, as illustrated in our motivating example (Section 5.1), each application lacks certain attributes, and thus the machine learning models of MLFM trained locally using each application's limited collection of attributes would not be accurate due to data unavailability. Therefore, V-FLFM utilizes Vertical Federated Learning (VFL) to enable the applications to collaboratively train a global ML model that can more accurately reflect the collection of all applications' attributes. Similar to HFL (Section 5.2.2), VFL allows the global model to be trained without the need for sharing data with other applications, with only a small amount of data and learning parameters exchanged with the central authority, thereby preserving confidentiality and privacy for the applications.

Similar to the H-FLFM approach, V-FLFM also has two stages: the *ML training stage* and the *ML application stage*, as depicted in Figure 5.6. What is unique to V-FLFM is the fact that the formal verification can only be performed at the central authority, where all attributes from the applications are collected. Specifically, during the *ML training stage*, the sampler identifies a small representative data sample, which is sent to the central authority for labeling (utilizing the *Verifier*). The labeled samples are then sent back to the corresponding applications and used by the *Learner* to train a global ML model. Each application's *Learner* calculates the learning parameters and sends them to the central authority for aggregation. The aggregated parameters are then returned to the applications to refine their local models. In the *ML application stage*, the final ML model of each application is applied to the remaining local audit data (i.e., the data not used during training) to prioritize verifying data records that are more likely to violate the property. Such data records are then sent to the central authority for verification utilizing the *Formal verifier*. The remainder of this section provides more details and examples for our V-FLFM methodology.

5.4.2 Training Stage - Training Data Building

V-FLFM requires the applications and the central authority to collaboratively obtain the training data and build the ML model (in contrast, H-FLFM does not involve the central authority for obtaining the training data). Specifically, as shown in Figure 5.6 (top-right inside *Application_1*), at the beginning of each iteration, the sampler interacts with the verifier (left inside *Central authority*) to build a training dataset. First, a sampling method is used to select a small representative data sample of fixed size from the audit data after applying the ML model built by the learner in the previous iteration (an initial ML model is provided for the first iteration). Second, since the data is vertically distributed, the necessary attributes for the verification are collected from the applications that hold them. The central authority

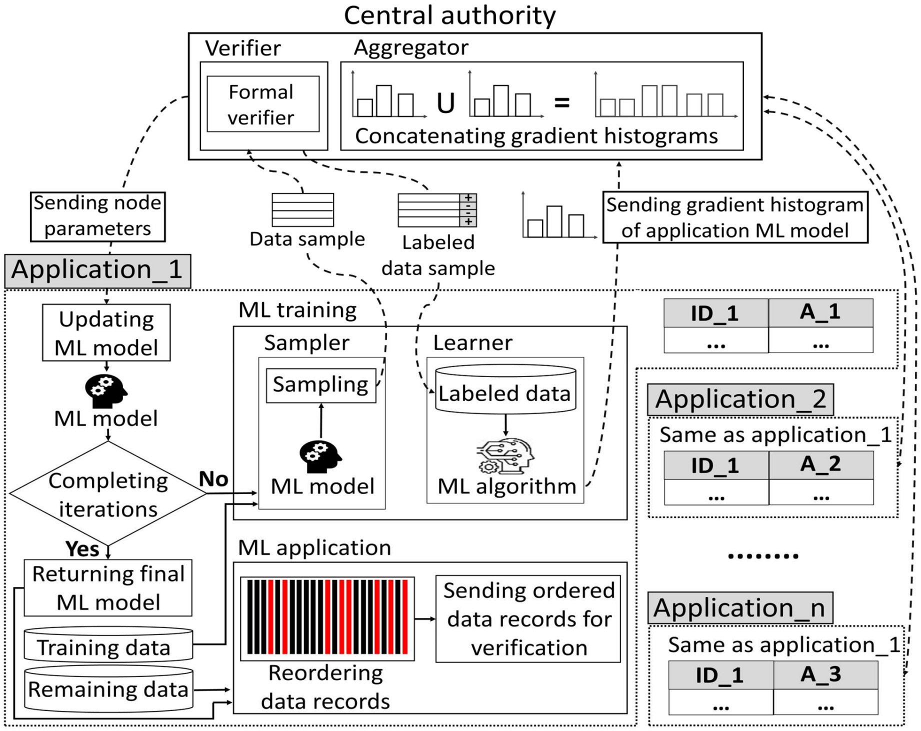


Figure 5.6: Overview of the vertical FLFM approach

then reassembles these samples into complete data records and verifies them using a formal verification approach. Thereafter, the data records labeled with verification results are divided into data samples and redistributed to the corresponding applications. The learner of each application then adds this newly received data sample to its existing training data (i.e., the collection of all previous samples) and utilizes this enriched training data to build a new ML model using a vertical federated learning algorithm.

Example 5.4.1 Figure 5.7 shows an example of the ML training over data vertically distributed between two applications, i.e., *Banking institution* and *Financial institution*. First, in Step ①, the sampler selects a sample of size ($m = 1$), using the uncertainty sampling method, which is assumed to include the data record (1) in this example, for both applications. Next, in Step ②, the sample from each application is sent to the *Verifier* inside the central authority (omitted on the *Financial institution*'s side for simplicity). The *Verifier* reassembles those samples into a complete data record, and then verifies the record and labels it based on the verification result (Step ③). Then, the labeled data record is divided into samples and sent back to the corresponding applications (Step ④), which utilize them to build a new ML model.

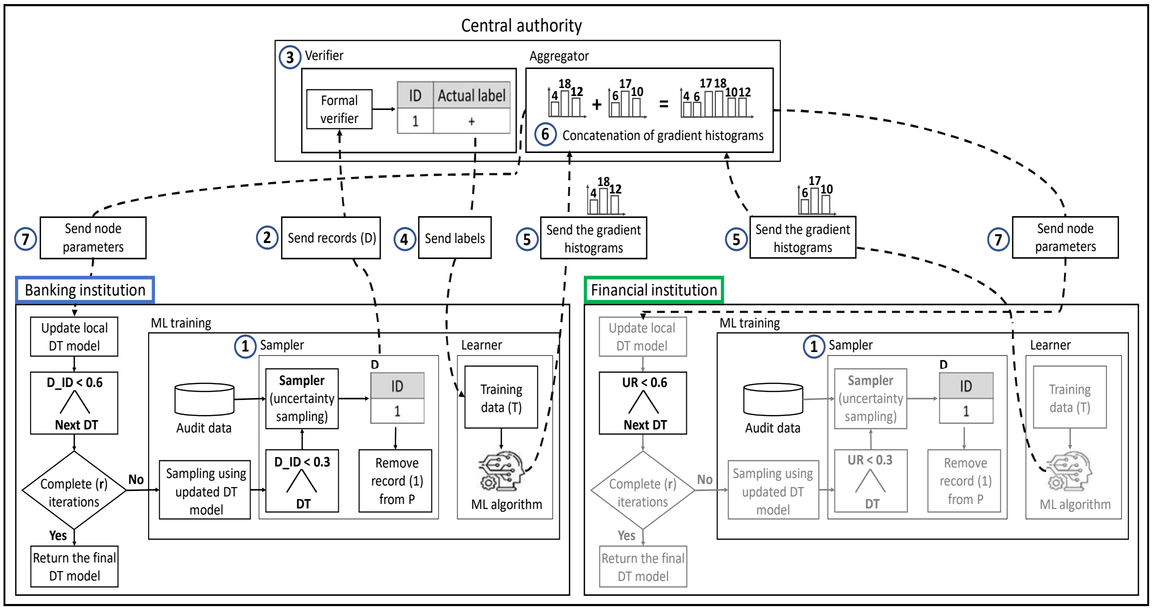


Figure 5.7: An example of the vertical FLFM training

5.4.3 Training Stage - Global Model Learning

The federated global model learning starts when each application receives the new labeled sample. Figure 5.6 (lower left inside *Application_1* and top) illustrates this using a vertical federated XGBoost classifier (detailed in Section 5.2.2). First, the VFL algorithm requires the application holding the labels to transmit them to the central authority, which then forwards these to the other applications for them to compute the corresponding gradients and hessians histograms (this step is performed only once for each model, as detailed in Section 5.2.2). Second, the applications send the histograms to the central authority for aggregation through concatenation (detailed in Section 5.2.2). The aggregated histograms are utilized to compute the node parameters, i.e., the best split or the prediction value if the current node is a leaf node, which are then sent back to the applications to update their local models. Finally, the updated local ML model is utilized by the `Sampler` to initiate the training of a new ML model. This training process may repeat for several iterations until a predefined stopping condition is met.

Example 5.4.2 Following Example 5.4.1, Figure 5.7 (Step ⑤) shows that the two applications send their learning parameters, i.e., $(4, 18, 12)$ and $(6, 17, 10)$ (only the gradient histograms are shown for clarity), to the central authority. The histograms (Step ⑥) are then concatenated by the *Aggregator* (inside the central authority), and used to compute the node parameters. These parameters are then sent back to the applications (Step ⑦) to update their local models. If the predefined number of iterations (r) has not been reached, the updated ML model is used in the next iteration of training; otherwise, it is passed to the ML application stage for utilization in subsequent verification process.

5.4.4 Application Stage

As Figure 5.6 (bottom of *Application_1*) shows, similar to H-FLFM, the final ML model obtained from the *ML training stage* is applied to the remaining audit data (i.e., the data not used for training) of each application to identify the “to be verified” subset (data records more likely to violate the given security property), which will be given a higher priority and sent first to the central authority for verification. On the other hand, the “not to be verified” subset will either be verified later or not verified at all, depending on the use cases (discussed in Section 5.6).

5.5 Implementation

This section details our implementation of FLFM.

Architecture. Figure 5.8 overviews our implementation of FLFM. Specifically, FLFM interacts with users to obtain the inputs including the security property to be verified and the system parameters, such as the number of iterations and the sample size (as detailed in Sections 5.3 and 5.4), and to deliver to users an audit report as the output. Furthermore, FLFM also interacts with the cloud environment hosting the microservice applications to acquire the necessary audit data for verification. The following details the components of FLFM.

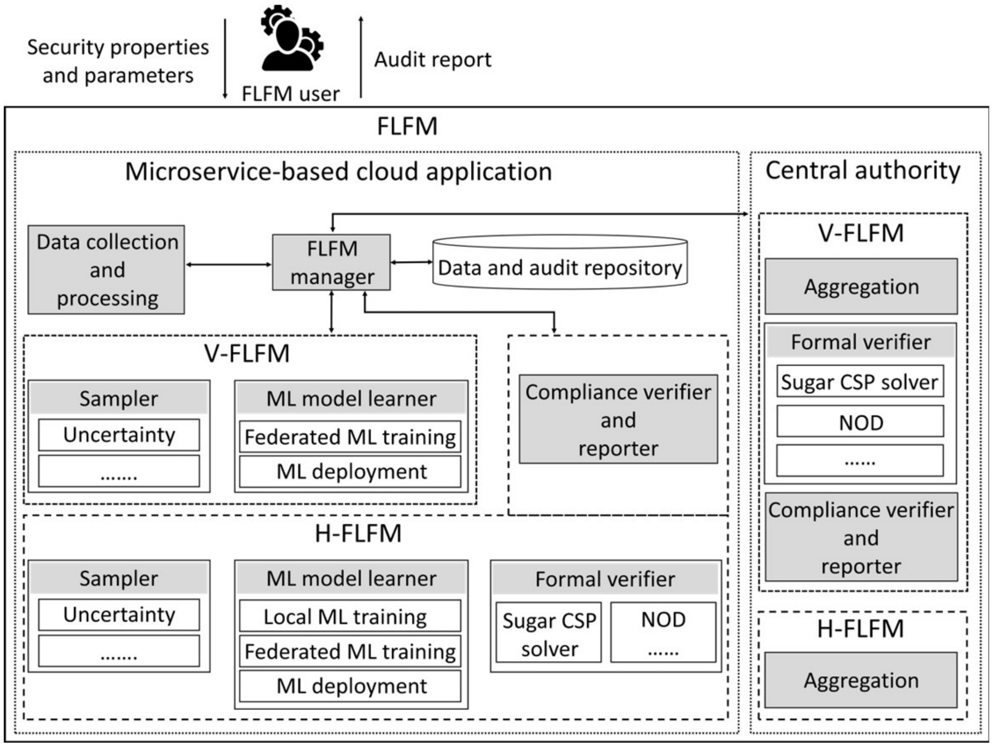


Figure 5.8: The FLFM architecture

Data Collection and Processing. This module (implemented in Python 3.6.9) collects audit data from the microservice-based applications and converts it into the necessary input formats for both the formal verifier and ML training.

FLFM Manager. This module (also implemented in Python 3.6.9) triggers either the horizontal or vertical FLFM approach based on the audit property and data distribution. Additionally, it manages and orchestrates the interactions among other modules for conducting data processing, data sampling, formal verification, ML training, etc., as detailed in Sections 5.3 and 5.4.

ML Model Learner. We utilize Python 3.6.9 and Scikit-learn 0.24.1 (an open source ML library written in Python) to implement this module. For both horizontal and vertical federated learning, we choose the XGBoost classifier [109], a scalable tree boosting algorithm that has seen wide application in real-world for its high accuracy and low false positive rate, with many awards in ML and data mining competitions [110, 111]. We employ *FedTree* [134], an FL system designed for tree-based models, to implement federated XGBoost. *FedTree* employs a histogram-sharing strategy for both horizontal and vertical FL. It facilitates distributed computing for practical FL deployment with configurable privacy techniques like Differential Privacy and Homomorphic Encryption, and supports standalone FL simulation on a single machine. Although many FL studies focus on neural networks, our work excludes them due to their widely recognized higher complexity and longer training time compared to traditional ML models [112], which is not aligned with our main objective of reducing the overall delay before violations can be identified.

Aggregation. This module (implemented in Python 3.6.9) aggregates the learning parameters received from the microservice-based applications.

Sampler. We utilize the *modAL* framework [113] to implement sampling strategies within this module. *modAL* is an active learning framework for Python3, built on top of Scikit-learn [114], which enables the rapid creation of active learning workflows with flexibility [113]. In our implementation, we opt for the uncertainty sampling strategy due to its superior computational efficiency compared to other strategies [46]. Moreover, our previous work [5] has demonstrated that combining uncertainty sampling with XGBoost achieves the highest verification performance.

Formal Verifier. We formalize the security properties along with the audit data as a Constraint Satisfaction Problem (CSP), a time-proven technique to express intricate problems.

Leveraging CSP permits users to articulate a wide range of security properties (owing to its expressiveness) in a relatively simple manner, since CSP can facilitate the uniform representation of audit data and security properties, presenting them in a clear and understandable formalism, such as first-order logic (FOL) [115]). As a future work, FLFM may also be integrated with other robust and efficient CSP solver algorithms, which can circumvent the costly state space traversal [116] to further enhance the scalability, particularly in larger environments.

After being formulated as a CSP problem, the security verification is performed using Sugar [67], a well-established, award-winning SAT-based constraint solver (e.g., the global constraint categories at the International CSP Solver Competitions in 2008 and 2009 [117]). Sugar tackles a finite linear CSP by converting it into a SAT problem through the order encoding method, and then it solves the SAT problem using the MiniSat solver [118], an efficient CDCL SAT solver recognized for its effectiveness in narrowing the search space [119]. A future direction is to adapt the FLFM framework to other verification methods (such as theorem proving, model checkers, temporal logic, and Datalog) based on the specific requirements of the verification tasks.

5.6 Experiment

This section first describes the investigated properties. Then it covers the experimental settings, use cases of our approach, and the datasets used. Finally, it presents our evaluation results for both H-FLFM and V-FLFM.

5.6.1 Investigated Properties

FLFM targets system-wide security properties that cannot be fully verified by analyzing the configuration of a single application or microservice in isolation. These properties depend on the relationships, communications, or configurations across the entire system, requiring the aggregation of information from multiple applications or services. These properties describe policies like communication isolation, role-based access compliance, and dependency integrity, which span multiple services and require system-wide visibility or coordination.

Our approach can support the verification of various security or custom properties as long as they can be verified using the chosen formal method tool. To make our discussion more concrete, we present two example properties (which will be used to evaluate our approach later in this section).

Rate Limiting Property. Ensures that applications adhere to predefined thresholds for request rates, such as a specified number of requests that are allowed within a given time frame. It serves as a security mechanism to protect networks by regulating traffic flow, preventing excessive rates that could lead to congestion or overload, and mitigating risks such as API abuse through excessive calls [135]. Additionally, it safeguards systems from malicious activities like Denial of Service (DoS) attacks [136] and ensures fair resource allocation among services. Therefore, verifying rate limits is crucial for maintaining the security, stability, and quality of service in cloud-based networking environments.

In our work, we assume that the cloud provider hosts several applications for different clients and enforces traffic rate limits on certain applications (e.g., no more than 1,000 requests per second for banking applications). Our objective is to compare the actual request rate against rate-limiting policies to ensure the compliance of these applications. Specifically we verify the following.

$$\forall \mathtt{id} \in \mathtt{AplicationID}, \forall \mathtt{r} \in \mathtt{Requests}, \forall \mathtt{t} \in \mathtt{time} : \mathtt{AppRequestCount}(\mathtt{id}, \mathtt{r}, \mathtt{t}) \\ <= \mathtt{RateLimit} \tag{3}$$

Where *AppRequestCount* denotes the number of requests *r* made by an application with ID *id* at a specific time interval ending at time *t*, and *RateLimit* is a constant representing the rate limit.

Access Control Property. In our work, we assume that two organizations (each with its own MS applications) work on a shared project. One of them is the project lead (say organization_A) hence it holds higher priority e.g., owns sensitive data for the project, or manages role assignments and access control (e.g., issuing permissions). On the other hand, the other organization (say organization_B) uses the data or shared resources provided by "organization_A" and maintains its own users access information. For instance, the two organizations could be a bank and financial institution that performs some tasks on behalf of the bank, such as risk assessment, credit scoring, or fraud detection.

Our objective is to ensure the compliance of established access control policies across these applications. For instance, one such policy might state: "Only users with the role *Analyst* from Organization_B are permitted to access *SharedResourceX*, and only if they are using a device that has been explicitly approved and registered by organization_A". Which would be represented as follows.

$$\begin{aligned}&\forall \texttt{id} \in \texttt{ResourceID}, \forall \texttt{dev} \in \texttt{DeviceID}, \forall \texttt{ur} \in \texttt{Roles} : \texttt{ResourceUserRole}(\texttt{id}, \texttt{ur}) \\ &\wedge (\texttt{ur} == \text{``}\texttt{Analyst}\text{''}) \wedge \texttt{UsedDeviceID}(\texttt{id}, \texttt{dev}) \wedge \texttt{ApprovedDevices}(\texttt{id}, \texttt{dev})\end{aligned} \tag{4}$$

Where *ResourceUserRole* represents the role (*ur*) of the user who accessed the *SharedResourceX* identified by its ID (*id*), *UsedDeviceID* represents the used device (*dev*) to access resource (*id*), and *ApprovedDevices* represent the approved devices by organization_A to access resource (*id*) from.

5.6.2 Experimental Settings and Datasets

Experimental Settings. All experiments are performed on a SuperServer 6029P-WTR running the Ubuntu 18.04 operating system equipped with Intel(R) Xeon(R) Bronze 3104 CPU@1.70GHz and 128GB of RAM without GPUs. All the experiments are performed using Sugar [67] as the formal verifier, FedTree [134] standalone simulator to simulate the federated settings, and Python 3.6.9 with Scikit-learn 0.24.1 ML packages for the ML method (i.e., XGBoost classifier [109]). For all experiments, we use the default parameters for the ML models. Each experiment is repeated 1,000 times to obtain the average results.

We evaluate FLFM under two use cases, one for the shortest verification time and the other for more complete results. First, in the *partial verification* use case, FLFM will stop after verifying all the "to be verified" records (i.e., the "not to be verified" records will not be verified, as detailed in Sections 5.3.4 and 5.4.4). This use case may apply when the user wants to find violations as quickly as possible (but not necessarily all the violations). Second, the *priority-based verification* use case means that FLFM will not stop after verifying the "to be verified" records, but instead will continue to verify the remaining ("not to be verified") records. This use case applies when the user requires complete verification results (at the cost of a longer running time).

Finally, since our previous work (MLFM) demonstrated through experimental evaluations that combining machine learning with formal methods outperforms traditional formal methods, we did not directly compare FLFM with traditional approaches. Instead, we focus on evaluating the performance of FLFM and comparing it with MLFM, as FLFM represents a distributed extension of MLFM. Our goal is to build upon the results established by MLFM and assess how they extend to the federated FLFM setting.

Datasets. To evaluate the performance of FLFM in the horizontal case (H-FLFM), we generate six sets of datasets, with each set containing two datasets, one for each application. Each dataset contains 12,500 data records for verifying the *rate limiting* property (P1 henceforth). Since data heterogeneity is the key factor affecting the ML accuracy in the horizontal case, the six sets of datasets are designed to have an increasing level of heterogeneity, e.g., the last set has 100% heterogeneity, i.e., the two applications do not share any common data values. Finally, each dataset contains around 10% of (uniformly distributed) records that violate the corresponding property.

For the vertical case (V-FLFM), we also generate six sets of datasets for both applications. Each dataset contains 12,500 data records for verifying the *access control* property (P2 henceforth). Unlike the horizontal case (where data heterogeneity is the key factor), the ML accuracy in V-FLFM is mainly affected by the missing attributes in each application. Each subsequent set of datasets is designed to have 10% more records that result in false prediction by the ML model due to the lack of attributes (e.g., data records that are identified by ML as "to be verified" but do not violate the given property). Finally, similar to the horizontal case, each dataset also contains around 10% of (uniformly distributed) records that violate the corresponding property.

5.6.3 H-FLFM Experimental Results

We evaluate the H-FLFM performance and how it may be impacted by data heterogeneity and other parameters. We also compare the performance of H-FLFM with a state-of-the-art approach, MLFM [5], which runs at each application to construct an ML model and identify the data records that violate the given property based on the local data. Table 5.1 lists the parameters evaluated in those experiments.

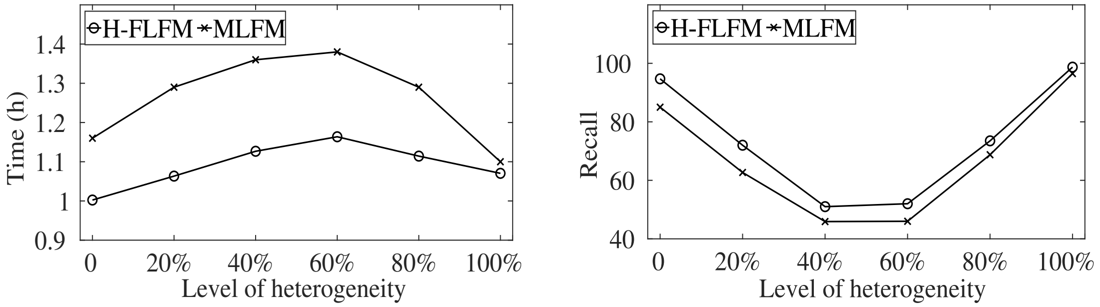
Parameter	Meaning	Abbreviation
Sample size	Number of records selected by the sampler from the audit data	**m**
Local iterations	Number of local iterations conducted between the teacher and the learner	**n**
Global (aggregation) iterations	Number of global iterations conducted between the applications and the central authority	**r**

Table 5.1: Main parameters evaluated in the experiments.

Impact of Data Heterogeneity. The first set of experiments evaluates the impact of different levels of data heterogeneity on the performance of H-FLFM in comparison to MLFM, with respect to the execution time and the recall, respectively. This experiment is performed using the best performing parameters of MLFM as reported in [5] (i.e., XGBoost with uncertainty sampling, 20% training data, sample size m = 250, and iteration count n = 10) for both MLFM and H-FLFM.

First, Figure 5.9 (a) shows the execution time for identifying violations (under the priority-based verification use case) for both H-FLFM and MLFM on datasets with different levels of data heterogeneity (ranging from 0% to 100%). The results show that H-FLFM is faster than MLFM across all levels of data heterogeneity, with an average of around 13% better performance for H-FLFM. This is expected as the federated global model learning of H-FLFM allows the applications to build more accurate ML models and consequently to prioritize the verification of data records more effectively than MLFM does. The results also show a similar trend for both approaches, i.e., as the level of data heterogeneity increases, the execution time first increases and then decreases passing around 60% data heterogeneity. This can be explained by a similar trend in the accuracy of the ML models, as detailed below.

Second, Figure 5.9 (b) shows the recall values for both H-FLFM and MLFM under different levels of data heterogeneity. The results show that H-FLFM achieves higher recall values (i.e., more effective in identifying the violations) than MLFM across all levels of heterogeneity. The results also show a similar trend for both approaches, i.e., as the level of data heterogeneity increases, the recall values first decrease and then increase. This can be explained by the fact that a medium level of heterogeneity (e.g., 40%) means the local data of an application contains an imbalanced mixture of data from different ranges, which makes the ML training more challenging; conversely, the ML training is easier either at a very low level of heterogeneity since the local data has a well balanced mixture of data from different ranges, or at a very high level of heterogeneity since the local data is mostly from the same range. This also explains the diminishing difference between H-FLFM and MLFM as the heterogeneity approaches 100% (i.e., the local model is good enough on local data from the same range). In the subsequent experiments, we choose datasets with a high heterogeneity level (i.e., 80%) since this is the most challenging case for H-FLFM (i.e., with the minimum benefit compared to MLFM).



(a) (b)

Figure 5.9: The execution time (in hours) (a) and recall (b) for identifying violations in datasets with different levels of heterogeneity by H-FLFM and by MLFM [5]

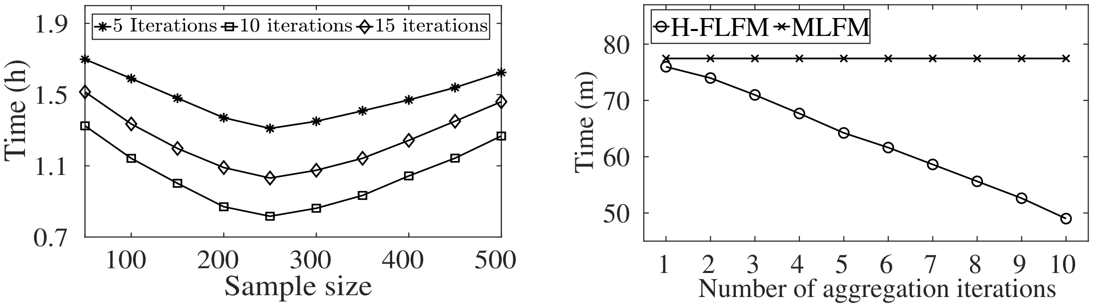
Impact of Parameters. The goal of this set of experiments is to study the impact of the H-FLFM parameters as listed in Table 5.1. First, Figure 5.10 (a) shows the effect of the sample size under various numbers of both the local iteration (n) and global (aggregation) iteration (r), where we choose $n = r$ (i.e., each local iteration is followed by an aggregation).

With respect to the effect of sample size, the results show that, with the sample size increasing (i.e., more training data is employed during each iteration) from 0 to 250 records, the H-FLFM performance improves owing to the enhanced accuracy of the ML model, where the lowest execution time for H-FLFM is achieved when the sample size reaches around 250, independently of the number of iterations. The improvement starts to diminish passing 250 records, since with larger sample sizes, the verification time during the training stage becomes more dominant (compared to the time saved during the application stage), leading to overall a longer time.

With respect to the effect of the number of iterations, the results show that, for any given sample size, the execution time of H-FLFM is the lowest under a medium number of iterations (e.g., 10 iterations), while a smaller number (e.g., 5 iterations) leads to the highest execution time. This can be explained by the fact that the increase in the number of iterations initially leads to more accurate ML models (as more training data is used) and consequently a lower execution time, while further increase in the number of iterations reverses the trend, since the accuracy improvement will diminish and the increase in training time due to the additional iterations will outweigh the time saved in the ML application stage. Therefore, we can conclude with the optimal H-FLFM parameters of ($m = 250$, $n = 10$, and $r = 10$).

Figure 5.10 (b) further studies the effect of the number of aggregation iterations on the execution time (for priority-based verification) of H-FLFM in comparison to MLFM, using the dataset of 80% heterogeneity. Here we fix the sample size and the number of local iterations at the best performing parameters of H-FLFM, i.e., (m = 250 and n = 10), while varying the number of aggregation iterations by performing the aggregation after multiple local iterations. The results show that increasing the number of aggregation iterations can significantly reduce the execution time, e.g., from 76 minutes for 1 aggregation to less than 50 minutes for 10 aggregations (i.e., aggregation is performed after every local iteration). The results also show that H-FLFM is always faster than MLFM, and the difference grows almost linearly in the number of aggregations, e.g., H-FLFM is 36.7% faster than MLFM with 10 aggregation iterations.

Performance and Robustness. The goal of this set of experiments is to further study the performance of H-FLFM in terms of execution time in comparison to MLFM, and the robustness in handling new data with different characteristics. First, Figure 5.11 (a) compares the execution time of H-FLFM to MLFM that is needed to identify different percentages of violation (for the priority-based verification) using the dataset of 80% heterogeneity and

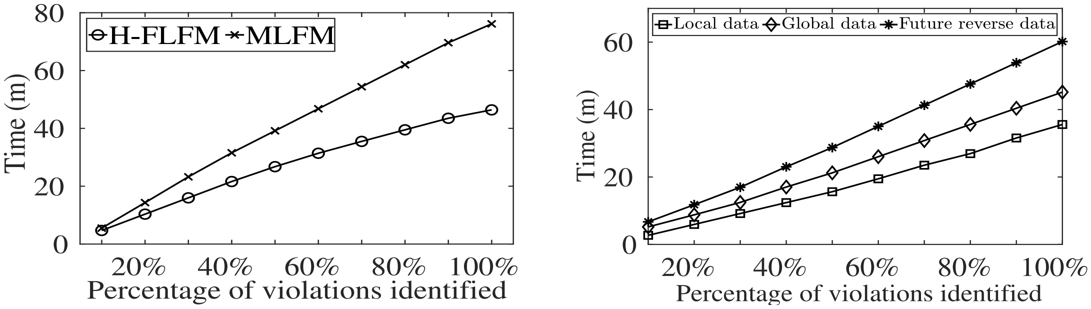


(a) (b)

Figure 5.10: The execution time of H-FLFM (in hours) for different sample sizes and for various numbers of iterations (aggregation and local iterations) (a). Execution time (in minutes) of H-FLFM using the best performing parameters (i.e., m = 250 and n = 10), while varying the number of aggregation iterations and compared to the execution time of MLFM using the same applicable parameters (b)

the best performing parameters (i.e., m = 250, n = 10, and r = 10). The results show that H-FLFM is always faster than MLFM in identifying violations, and H-FLFM takes around 39% less time than MLFM for identifying all the violations.

Second, Figure 5.11 (b) studies the robustness of H-FLFM and the generality of its ML model for handling new data records that have different distributions or data ranges from the local data. Specifically, *Local data* in the figure refers to the normal scenario used in previous experiments, i.e., H-FLFM applies its trained ML model to data records with similar distributions and value ranges as the training data. In addition, we test H-FLFM under two more challenging scenarios, i.e., *Future reverse data* which means the ML model trained at one application is applied to data records with distributions and value ranges similar to the other application, and *Global data* which means the ML model is applied to data records that have mixed distributions and value ranges from both applications. As expected, H-FLFM performs the best for the local data scenario. The performance becomes worse under the *Global data* scenario, and decreases further under the *Future reverse data* scenario. Nonetheless, even in the last case, by taking approximately 60 minutes, H-FLFM is still around 21% faster than MLFM (depicted in Figure 5.11 (a), which requires around 76 minutes).



(a) (b)

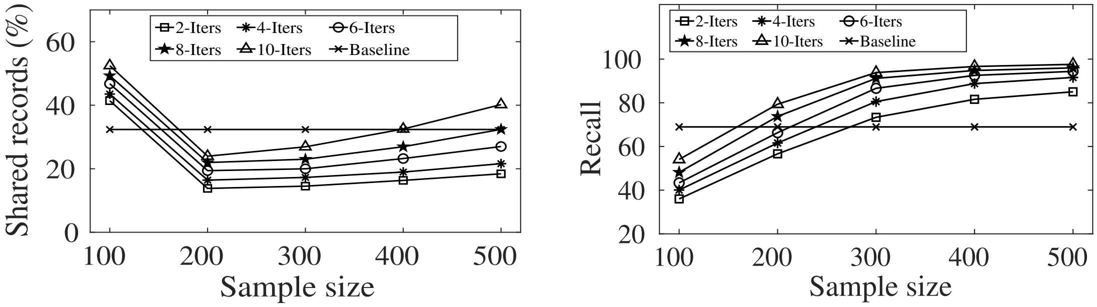
Figure 5.11: The execution time (in minutes) for identifying different percentages of violations by H-FLFM and MLFM (a). The time (in minutes) for identifying different percentages of violations by H-FLFM under different scenarios (b)

5.6.4 V-FLFM Experimental Results

We evaluate the V-FLFM performance and how it may be impacted by different parameters. We also compare the performance of V-FLFM with the baseline approach (i.e., applying MLFM on the local data, without the federated global learning step of V-FLFM).

Impact of Parameters. As mentioned in Section 5.4, under the vertical case of FLFM (V-FLFM), the formal verification is performed at the central authority by collecting all the attributes from the applications. Therefore, one aspect of the V-FLFM performance is the percentage of data records that need to be shared with the central authority. Specifically, Figure 5.12 (a) shows the impact of the V-FLFM parameters (i.e., the sample size and the number of iterations) on the percentage of shared records for the partial verification case. It also compares V-FLFM to the baseline approach using the best performing parameters of MLFM as reported in [5] (i.e., XGBoost with uncertainty sampling, 20% training data, sample size m = 250, and iteration count n = 10). The results show that V-FLFM performs better than the baseline (in terms of a lower percentage of shared records) in most cases. For V-FLFM, the percentage of shared records initially decreases until the sample size reaches around 200. This shows that a larger sample size allows V-FLFM to be more effective in identifying violations due to the increasing accuracy of its ML model. As the sample size further increases, the trend reverses, i.e., the percentage of shared records also starts to increase, since the improvement in model accuracy diminishes while the increase in the percentage of shared records due to larger samples becomes more dominant. The results also show that a larger number of iterations generally means more shared records, and hence a single iteration (with the sample size of (200)) would yield the lowest percentage of shared records.

Figure 5.12 (b) shows the recall values of both V-FLFM and the baseline approach. The results show that V-FLFM achieves higher recall with more iterations, since the accuracy of the ML model generally increases with more iterations of training. The results also show that different combinations of the sample size and number of iterations can achieve similar recall values. Therefore, the best combination of the parameters could depend on the users' preferences (e.g., less shared records or higher recall). In our experiments, we choose the sample size of (300) and iteration count of (8) as the optimal parameters, which means our approach shares 28.92% fewer records compared to the baseline approach, while achieving a 32.25% higher recall compared to the baseline approach.

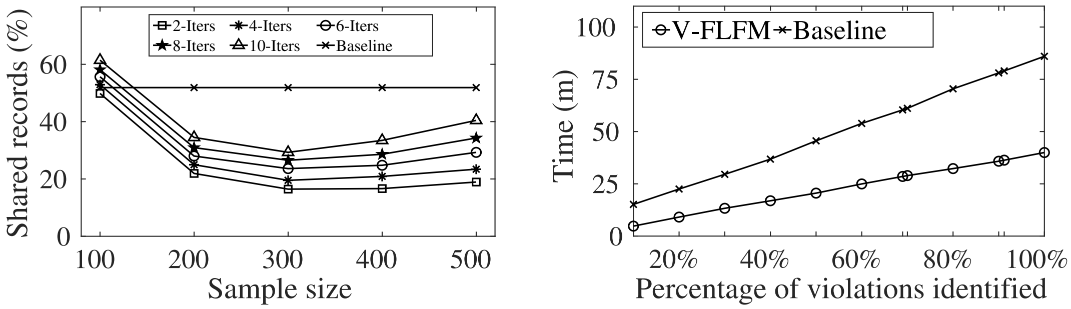


(a)

(b)

Figure 5.12: The percentage of records shared with the central authority by V-FLFM and the baseline approach for the partial verification case (a) and the recall of V-FLFM and the baseline approach (b)

Figure 5.13 (a) shows the percentage of records that are shared by V-FLFM and the baseline approach for the priority-based verification (note that the count of records is taken as soon as all the violations are identified). The results show that both V-FLFM and the baseline approach lead to more shared records than in the partial verification case (due to the need for verifying the “not to be verified” records). V-FLFM shows a similar trend as before, i.e., the percentage of shared records initially decreases due to more accurate ML models (with the lowest value of around (16.45%) under the sample size of (300) and the number of iterations of (1)), and then the trend reverses as the increase in shared records due to larger samples becomes more dominant.

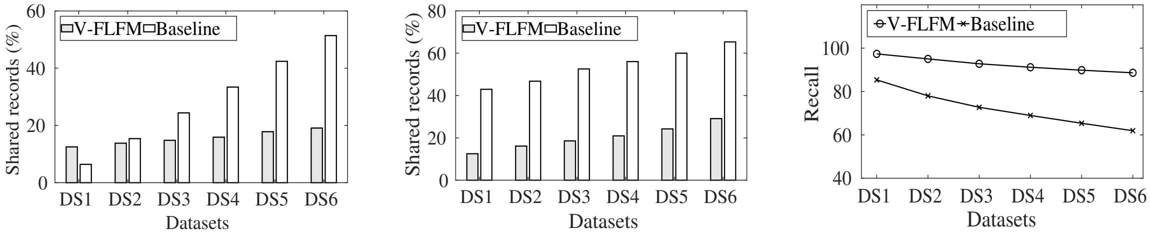


(a) (b)

Figure 5.13: The percentage of records shared with the central authority by V-FLFM and the baseline approach for the priority-based verification case (a) and the execution time (in minutes) for identifying different percentages of violations by V-FLFM and the baseline approach (b)

Performance. The goal of this set of experiments is to further study the performance of V-FLFM using the best performing parameters in terms of both the execution time and shared records in comparison to the baseline approach. First, Figure 5.13 (b) compares the execution time of V-FLFM to the baseline approach for identifying different Percentages of violations (for the priority-based verification) using the best performing parameters (i.e., m =300 and r = 8). The results show that V-FLFM outperforms the baseline in all cases, e.g., V-FLFM can identify all the violations in around 40 minutes, which takes the baseline around 86 minutes, i.e., V-FLFM takes around 53.52% less time to identify the same percentage of violations.

Second, Figure 5.14 compares the percentage of shared records by V-FLFM and the baseline approach using the best performing parameters (i.e., m =300 and r = 8) on six different datasets. As mentioned in Section 5.6.2, each subsequent dataset contains 10% more records that are mistakenly identified by ML as “to be verified”, i.e., false violations (while all the datasets contain 10% of true violations). Figure 5.14 (a) shows that the increase in false violations affects the baseline approach more than V-FLFM for the partial verification, and V-FLFM outperforms the baseline approach on all the datasets (on average, V-FLFM shares almost 28.42% less records than the baseline approach). Figure 5.14 (b) shows similar results for the priority-based verification case. Although both approaches lead to more shared records in this case, V-FLFM shares almost 45.45% less records on average than the baseline approach over all the datasets. Finally, Figure 5.14 (c) shows that the recall values of V-FLFM decreases when the number of false violations increases (which cause the ML model to become less accurate) although it remains to be higher than the baseline approach.



(a) (b) (c)

Figure 5.14: Percentage of shared records by V-FLFM and the baseline approach for partial verification (a) and priority-based verification (b) using different datasets, and using different datasets, and the recall of V-FLFM and the baseline approach using different datasets (c)

5.7 Summary

We have presented FLFM, a novel approach designed to more quickly identify security violations for microservice applications. FLFM combines the efficiency and privacy-friendliness of Federated Learning (FL) with the rigor of formal methods (FM). In particular, during the training stage, each microservice application first selects a small yet representative sample of its configuration data and labels it according to the formal method verification results obtained for that sample. Then, the applications leverage federated learning to jointly train a global ML model based on the collection of their samples. In the application stage, this global ML model is applied to help the applications prioritize more suspicious data for earlier verification. We provided detailed methodologies for both the horizontal and vertical FL cases, and our experimental results demonstrated that FLFM outperforms the baseline approach.

Chapter 6

Conclusion

In this thesis, we addressed key security verification challenges in modern virtualized environments by introducing three novel solutions tailored to the complexity, scalability, and privacy concerns of NFV and microservice architectures. First, we introduced NFV-Guard+, a cross-level formal verification solution that validates security properties across the entire NFV stack without the need for explicit verification at each level. Second, to improve the efficiency of existing security verification solutions, particularly formal methods, we developed the MLFM approach, which integrates the speed of machine learning with the rigor of formal methods to accelerate and prioritize verification without compromising accuracy. Finally, we proposed FLFM, a privacy-preserving, federated learning–guided formal security verification solution designed for the distributed nature of microservices. The following discusses the limitations and our future research focus:

1. First, although our cross-level security verification approach for NFV is platform-agnostic, the current implementation of data collection and processing is limited to OpenStack/Tacker. To address this limitation, future work will focus on a more modular design with a clear methodology for extending support to other open-source NFV platforms, such as OPNFV and OSM.

2. Second, our machine learning-guided formal method for faster identification of security breaches is currently limited to NFV environments. A future direction is to extend MLFM to other large-scale virtual infrastructures, such as cloud platforms and SDNs. Additionally, while MLFM focuses solely on security verification, an important next step is to integrate it with security enforcement mechanisms to enable faster attack prevention. Furthermore, MLFM currently operates in a static, on-demand manner using data snapshots, and future work will explore continuous security verification through real-time monitoring with data streams.

3. Third, for our security verification of microservices using federated learning-guided formal methods, there are several potential future directions. One promising avenue is the integration of privacy-preserving mechanisms, such as secure multi-party computation or differential privacy, to further strengthen security while ensuring data confidentiality. Additionally, improving FLFM's adaptability by incorporating adaptive federated learning techniques could enhance its efficiency in addressing evolving security threats. Moreover, in real-world scenarios, data distribution is rarely perfectly horizontal or vertical; both types may coexist. Developing hybrid FL solution capable of handling both types of partitioning simultaneously represents another important direction for our future research.

Bibliography

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

[2] OpenStack, “OpenStack,” 2020, available at: https://www.openstack.org/.

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

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

[5] A. Oqaily, Y. Jarraya, L. Wang, M. Pourzandi, and S. Majumdar, “Mlfm: Machine learning meets formal method for faster identification of security breaches in network functions virtualization (nfv),” in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 466–489.

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

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

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

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

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

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

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

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

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

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

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

[17] 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.

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

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

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

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

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

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

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

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

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

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

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

[29] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi,

L. Wang, and M. Debbabi, “TenantGuard: Scalable runtime verification of cloud-wide VM-level network isolation,” in *NDSS*, 2017.

[30] T. Madi, Y. Jarraya, A. Alimohammadifar, S. Majumdar, Y. Wang, M. Pourzandi, L. Wang, and M. Debbabi, “ISOTOP: Auditing virtual networks isolation across cloud layers in OpenStack,” *ACM TOPS*, vol. 22, no. 1, pp. 1:1–1:35, 2018.

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

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

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

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

[35] G. Marchetto, R. Sisto, J. Yusupov, and A. Ksentini, “Virtual network embedding with formal reachability assurance,” in *14th International Conference on Network and Service Management*, 2018, pp. 368–372.

[36] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, “Formal verification of virtual network function graphs in an sp-devops context,” in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2015, pp. 253–262.

[37] 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)*, vol. 22, no. 1, pp. 1–35, 2018.

[38] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang, "Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack," in *ACM CODASPY*, 2016.

[39] S. Majumdar, T. Madi, Y. Wang, Y. Jarraya, M. Pourzandi, L. Wang, and M. Debbabi, "User-level runtime security auditing for the cloud," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1185–1199, 2017.

[40] S. Majumdar, Y. Jarraya, T. Madi, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Proactive verification of security compliance for clouds through pre-computation: Application to OpenStack," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 47–66.

[41] S. Majumdar, Y. Jarraya, M. Oqaily, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Leaps: Learning-based proactive security auditing for clouds," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 265–285.

[42] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ice learning for synthesizing invariants and contracts," *In: Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[43] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 69–87.

[44] S. Ren and X. Zhang, "Synthesizing conjunctive and disjunctive linear invariants by K-means++ and SVM." *Int. Arab J. Inf. Technol.*, vol. 17, no. 6, pp. 847–856, 2020.

[45] Y. Vizel, A. Gurfinkel, S. Shoham, and S. Malik, “IC3-flipping the E in ICE,” in *International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 2017, pp. 521–538.

[46] B. Settles, “Active learning literature survey,” 2009.

[47] A. Panda, M. Sagiv, and S. Shenker, “Verification in the age of microservices,” in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 30–36.

[48] X. Meng, X. Duan, W. Tao, Y. Luan, J. Zhang, and D. Wu, “Modeling and verification of industrial microservice architecture based on formal methods,” in *2021 China Automation Congress (CAC).* IEEE, 2021, pp. 3776–3780.

[49] F. Dai, H. Chen, Z. Qiang, Z. Liang, B. Huang, and L. Wang, “Automatic analysis of complex interactions in microservice systems,” *Complexity*, vol. 2020, pp. 1–12, 2020.

[50] M. M. Ghorbani, F. F. Moghaddam, M. Zhang, M. Pourzandi, K. K. Nguyen, and M. Cheriet, “Malchain: Virtual application behaviour profiling by aggregated microservice data exchange graph,” in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE, 2020, pp. 41–48.

[51] M.-O. Pahl and F.-X. Aubet, “All eyes on you: Distributed multi-dimensional iot microservice anomaly detection,” in *2018 14th International Conference on Network and Service Management (CNSM).* IEEE, 2018, pp. 72–80.

[52] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, “Automatic policy generation for {Inter-Service} access control of microservices,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3971–3988.

[53] C. Meadows, S. Hounsinou, T. Wood, and G. Bloom, "Sidecar-based path-aware security for microservices," in *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies*, 2023, pp. 157–162.

[54] A. Venčkauskas, D. Kukta, Š. Grigaliūnas, and R. Brūzgienė, "Enhancing microservices security with token-based access control method. 23 (6), 3363," 2023.

[55] M.-O. Pahl, F.-X. Aubet, and S. Liebald, "Graph-based iot microservice security," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–3.

[56] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "{CRISP}: Critical path analysis of {Large-Scale} microservice architectures," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 655–672.

[57] V. Mothukuri, P. Khare, R. M. Parizi, S. Pouriyeh, A. Dehghantanha, and G. Srivastava, "Federated-learning-based anomaly detection for iot security attacks," *IEEE Internet of Things Journal*, vol. 9, no. 4, pp. 2545–2554, 2021.

[58] Y. Zhao, J. Chen, D. Wu, J. Teng, and S. Yu, "Multi-task network anomaly detection using federated learning," in *Proceedings of the 10th international symposium on information and communication technology*, 2019, pp. 273–279.

[59] Y. Liu, S. Garg, J. Nie, Y. Zhang, Z. Xiong, J. Kang, and M. S. Hossain, "Deep anomaly detection for time-series data in industrial iot: A communication-efficient on-device federated learning approach," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6348–6358, 2020.

[60] D. C. Attota, V. Mothukuri, R. M. Parizi, and S. Pouriyeh, "An ensemble multi-view federated learning intrusion detection for iot," *IEEE Access*, vol. 9, pp. 117 734–117 745, 2021.

[61] Z. He, J. Yin, Y. Wang, G. Gui, B. Adebisi, T. Ohtsuki, H. Gacanin, and H. Sari, "Edge device identification based on federated learning and network traffic feature engineering," *IEEE Transactions on Cognitive Communications and Networking*, vol. 8, no. 4, pp. 1898–1909, 2021.

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

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

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

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

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

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

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

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

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

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

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

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

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

[75] SP, NIST, "800-53," *Recommended security controls for federal information systems*, pp. 800–53, 2003.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[95] OSM, “Open Source MANO,” 2022, available at: https://osm.etsi.org/.

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

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

[98] A. A. Barakabitze, A. Ahmad, R. Mijumbi, and A. Hines, "5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, p. 106984, 2020.

[99] Z. Kotulski, T. W. Nowak, M. Sepczuk, M. Tunia, R. Artych, K. Bocianiak, T. Osko, and J.-P. Wary, "Towards constructive approach to end-to-end slice isolation in 5G networks," *EURASIP Journal on Information Security*, vol. 2018, no. 1, pp. 1–23, 2018.

[100] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman, "Automated analysis and debugging of network connectivity policies," *Microsoft Research*, pp. 1–11, 2014.

[101] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review," *Computer Standards & Interfaces*, vol. 58, pp. 1–22, 2018.

[102] OpenStack Training Labs, "OpenStack Training Labs," available at: https://wiki.openstack.org/wiki/Documentation/training-labs.

[103] P. Quinn and T. Nadeau, "Rfc 7948, problem statement for service function chaining," *Internet Engineering Task Force (IETF), ed*, 2015.

[104] N. Schear, P. T. Cable II, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and Maintaining Trust in the Cloud," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, 2016, pp. 65–77.

[105] M. C. Monard and G. E. Batista, "Learmng with skewed class distrihutions," *Advances in Logic, Artificial Intelligence, and Robotics: LAPTEC*, vol. 85, no. 2002, p. 173, 2002.

[106] S. Buss and J. Nordström, "Proof complexity and sat solving," *Handbook of Satisfiability*, vol. 336, pp. 233–350, 2021.

[107] H. M. Sani, C. Lei, and D. Neagu, "Computational complexity analysis of decision tree algorithms," in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2018, pp. 191–197.

[108] A. E. Mohamed, "Comparative study of four supervised machine learning techniques for classification," *Information Journal of applied science and technology*, vol. 7, no. 2, 2017.

[109] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[110] Z. Chen, F. Jiang, Y. Cheng, X. Gu, W. Liu, and J. Peng, "XGBoost classifier for DDoS attack detection and analysis in SDN-based cloud," in *IEEE international conference on big data and smart computing (BigComp)*, 2018, pp. 251–256.

[111] F. Neutatz, M. Mahdavi, and Z. Abedjan, "Ed2: A case for active learning in error detection," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 2249–2252.

[112] P. Maji and R. Mullins, "On the reduction of computational complexity of deep convolutional neural networks," *Entropy*, vol. 20, no. 4, p. 305, 2018.

[113] T. Danka and P. Horvath, "modAL: A modular active learning framework for Python," *arXiv preprint arXiv:1805.00979*, 2018.

[114] O. Kramer, "Scikit-learn," in *Machine learning for evolution strategies*. Springer, 2016, pp. 45–53.

[115] M. Ben-Ari, *Mathematical logic for computer science*. Springer Science & Business Media, 2012.

[116] I. Sassi, S. Anter, and A. Bekkhoucha, “A graph-based big data optimization approach using hidden markov model and constraint satisfaction problem,” *Journal of Big Data*, vol. 8, no. 1, pp. 1–29, 2021.

[117] “Sugar: a SAT-based Constraint Solver,” https://cspsat.gitlab.io/sugar/. Last accessed 8 November 2021.

[118] N. Eén and N. Sörensson, “An extensible sat-solver,” in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

[119] W. Gong and X. Zhou, “A survey of sat solver,” in *Proceedings of AIP Conference*, vol. 1836, no. 1. AIP Publishing LLC, 2017, p. 020059.

[120] Open Baton, “Open Baton,” 2017, https://openbaton.github.io/. Last accessed 16 June 2022.

[121] OPNFV, “Open Platform for NFV,” 2018, available at:https://www.opnfv.org/.

[122] D. S. Linthicum, “Practical use of microservices in moving workloads to the cloud,” *IEEE Cloud Computing*, vol. 3, no. 5, pp. 6–9, 2016.

[123] T. Hoff, “Lessons learned from scaling uber to 2000 engineers, 1000 services, and 8000 git repositories,” available at: https://goo.gl/1MRvoT. Last accessed 6 November 2023.

[124] M. Benedict and V. Charanya, “How we built a metering and chargeback system to incentivize higher resource utilization of twitter infrastructure,” available at: http://bit.ly/3aETlqs. Last accessed 6 November 2023.

[125] T. MAURO, “Adopting Microservices at Netflix: Lessons for Architectural Design,” available at: https://goo.gl/DyrtvI. Last accessed 6 November 2023.

[126] L. Li, Y. Fan, M. Tse, and K.-Y. Lin, "A review of applications in federated learning," *Computers & Industrial Engineering*, vol. 149, p. 106854, 2020.

[127] G. D. P. Regulation, "General data protection regulation (gdpr)," *Intersoft Consulting, Accessed in October*, vol. 24, no. 1, 2018.

[128] P. F. Edemekong, P. Annamaraju, and M. J. Haydel, "Health insurance portability and accountability act," 2018.

[129] E. Goldman, "An introduction to the california consumer privacy act (ccpa)," *Santa Clara Univ. Legal Studies Research Paper*, 2020.

[130] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[131] K. Cheng, T. Fan, Y. Jin, Y. Liu, T. Chen, D. Papadopoulos, and Q. Yang, "Secureboost: A lossless federated learning framework," *IEEE Intelligent Systems*, vol. 36, no. 6, pp. 87–98, 2021.

[132] Z. Tian, R. Zhang, X. Hou, J. Liu, and K. Ren, "Federboost: Private federated learning for gbdt," *arXiv preprint arXiv:2011.02796*, 2020.

[133] Y. J. Ong, Y. Zhou, N. Baracaldo, and H. Ludwig, "Adaptive histogram-based gradient boosted trees for federated learning," *arXiv preprint arXiv:2012.06670*, 2020.

[134] Q. Li, Z. Wu, Y. Cai, Y. Han, C. M. Yung, T. Fu, and B. He, "Fedtree: A federated learning system for trees," in *Proceedings of Machine Learning and Systems*, 2023.

[135] A. El Malki, U. Zdun, and C. Pautasso, "Impact of api rate limit on reliability of microservices-based architectures," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 19–28.

[136] J. Castro, N. Laranjeiro, and M. Vieira, “Detecting dos attacks in microservice applications: Approach and case study,” in *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, 2022, pp. 73–78.