MULTI-OBJECTIVE NETWORK VIRTUALIZATION AND ITS APPLICABILITY TO INDUSTRIAL NETWORKS

WASEEM MANDARAWI

A Thesis submitted for
Doctoral Degree

Chair of Computer Networks and Computer Communications
Faculty of Computer Science and Mathematics
University of Passau

April 14, 2022

Waseem Mandarawi : *Multi-objective Network Virtualization and its Applicability to Industrial Networks,* © April 14, 2022

REVIEWERS:
Prof. Dr.-Ing. Hermann de Meer
Professor of Computer Networks and Computer Communications
Universität Passau
Innstraße 43
94032 Passau, Germany
demeer@uni-passau.de
http://www.net.fim.uni-passau.de/

Prof. Dr.-Ing. Wolfgang Kellerer
Professor of Communication Networks
Technische Universität München
Arcisstrasse 21
80333 München, Germany
wolfgang.kellerer@tum.de
http://www.lkn.ei.tum.de/

ABSTRACT

Network virtualization provides high flexibility for deploying communication services in dense and heterogeneous environments. Two main approaches (dimensions) that are usually combined exist: Network Function Virtualization (NFV) technologies for functionality virtualization and Virtual Network Embedding (VNE) algorithms for resource virtualization. These approaches can be applied to different network levels, such as factory and enterprise levels of industrial networks. Several objectives and constraints, that might be conflicting, shall be considered when network virtualization is applied, mainly in complex topologies. This thesis proposes a network virtualization model that considers both virtualization dimensions, two network levels, and different objectives and constraints. The network levels considered are two primary levels in industrial networks. However, this consideration does not restrict the model to a particular environment or certain levels. The considered objectivities/constraints are topology, reliability, security, performance, and resource usage.

Based on this model, we first build an overall combined solution for autonomic and composite virtual networking. This solution considers both virtualization dimensions, two network levels, and target objectives. Furthermore, this solution combines three novel virtualization sub-approaches that consider performance, reliability, and performance. However, the sub-approaches apply to different combinations of levels and dimensions, and the reliability approach additionally considers the resource usage objective. After presenting all solutions, we map them to the defined model.

Regarding applicability to industrial networks, the combined approach is applied to an enterprise-level Industrial Internet of Things (IIoT) use case inspired by the smart factory concept in Industry 4.0. However, the sub-approaches are applied to more specific use cases. The performance and reliability solutions are integrated with relevant components of the Time Sensitive Networks (TSN) standard as a modern technology for industrial networks. The goal is to enrich the reliability and performance capabilities of TSN with the flexibility of network virtualization.

In the combined approach, we compose and embed an environment-aware Extended Virtual Network (EVN) that represents the physical devices, virtual application functions, and required Service Function Chains (SFCs). We use the graph transformation method to transform abstract application requirements (represented by an Application Request (AR)) into an EVN. Both EVN composition and embedding methods consider the Substrate Network (SN) topology and different security, reliability, performance, and resource usage policies. These policies are applied with a certain priority and depend on the properties of communicating entities such as location and type. The EVN is embedded using property-based node mapping, reliability-aware branching, and a greedy chain embedding heuristic. The chain embedding heuristic is evaluated using a random topology that represents the use case.

The performance sub-approach is NFV-based and is applied to a specific use case with Time-critical Traffic (TCT) flows. We develop and evaluate a complete framework for virtualizing Time-aware Shaper (TAS) using high-performance NFV. The reliability sub-approach is VNE-based and is applied to a specific factory level use case. We develop

minimal and maximal branching heuristics based on a reliability-aware k-shortest path algorithm and compare them using a typical factory topology. We then integrate these algorithms with a Frame Replication and Elimination for Reliability (FRER) simulator to realize reliability policies by the autonomic and efficient configuration of a supporting technology.

The security sub-approaches are related to both virtualization dimensions and are applied to generic enterprise-level use cases. However, the applicability of the security aspect to industrial networks is only shown in the combined (EVN) approach and its use case. We research the autonomic security management in Network Function Virtualization Infrastructure (NFVI) with the main goal of early reaction to threats through SFC reconfiguration through Virtual Network Function (VNF) live migration. This goal is approached by supporting the security measurements with a decision making architecture that considers, on the one hand, the threats and events in the environment and, on the other hand, the Service Level Agreement (SLA) between the NFVI provider and user. For this purpose, we classify the VNF-specific attacks and define possible early detectable behavior patterns. Finally, we develop a security-aware VNE heuristic that considers the security requirements of the Virtual Network (VN) and the security capabilities of the SN. This approach is modified in the combined approach to consider deploying virtualized security VNFs.

ACKNOWLEDGMENTS

I would like to sincerely thank my supervisor Prof. Hermann de Meer for all the discussions and support, and Prof. Wolfgang Kellerer for his efforts as a second reviewer.

The research on EVN composition and embedding, virtual TSN, and integration with FRER was supported by the German BMBF within the project FIND (Future Industrial Network Architecture).

The reliability-aware branching concept, the evaluation framework, and factory topology were developed in cooperation with Siemens.

The research on cloud security and security-aware VNE was supported by the Bavarian research association FORSEC (Security of highly networked IT systems).

Special thanks to Prof. Andreas Fischer for the support in the research on cloud security and security-aware VNE, and Eva Weishaeupl for the relevant collaboration on SLAs.

Many thanks to the partners of the project FORSEC, in particular, Prof. Hans P. Reiser for the cooperation on cloud security. Many thanks to FIND partners, in particular Siemens, for the cooperation on industrial networks, mainly TSN. These collaborations helped to build the required experience for conducting the research related to this thesis.

Special thanks to Ali Alshawish and Ammar Alyousef for the discussions on formalities, and the students Jürgen Rottmeier, Hamza Chahed, and Medina Saracevic, for the valuable cooperations.

Finally, thanks from the heart to my wife Dareen and children Adnan and Ragad, and to my big family (father, mother, brothers and sisters) for their great support and patience.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ALGORITHMS

ALEVIN	ALgorithms for Embedding of VIrtual Networks - a framework to develop, compare, and analyze VNE algorithms.
AR	Application Request - An abstraction of the application requirements that defines end-nodes and simplified performance, security, and reliability demands.
ARa	Acceptance Ratio - The ratio of the successfully embedded VNRs.
BET	Best Effort Traffic - The traffic flows without strict timing requirements.
CDF	Cumulative Distribution Function - The percentage of results within a given range (smaller than the given value).
CDL	Cross-domain Link - A VLi that passes through two different security domains.
DPDK	Data Plane Development Kit - A set of C libraries used to overcome the virtualization overhead.
EVN	Extended Virtual Network - A VN that results from extending the AR with SFCs, additional VNos and VLis, and the required physical devices.
FG	Forwarding Graph - A graph that defines the order of VNFs in the SFC.
FRER	Frame Replication and Elimination for Reliability - A TSN sub-standard for network resilience.
GCL	Gate Control List - A TAS-related structure attached to each port to define the gate opening times in which the TCT is transmitted.
IAT	Inter-Arrival Time - Time between successive frames transmitted by a traffic generator.

MaxBr	Maximal Branching - The maximal branching algorithm that uses the reliability-aware k-shortest paths algorithm.
MinBr	Minimal Branching - The minimal branching algorithm that uses the reliability-aware k-shortest paths algorithm.
NF	Network Function - A networking procedure or protocol that executes a specific service.
NFV	Network Function Virtualization - Executing network functions in a virtualized form using VMs or containers running on standard servers.
NFVI	Network Function Virtualization Infrastructure - The IaaS cloud environment hosting the SFCs.
PTP	Precision Time Protocol - A time synchronization protocol with high accuracy.
RSP	Reliability Shortest Path - Reliability-aware k-shortest paths algorithm.
R-TAG	Redundancy Tag - A sequence number encoding method in FRER.
RU	Resource Utilization - The degree of efficiency in using SN resources for embedding a VN.
SCL	Sending Control List - The list of TCT transmission times at the talker (for TAS).
SFC	Service Function Chain - A sequence of VNFs that provide an end-to-end network service.
SLA	Service Level Agreement - A set of metrics related to service quality agreed on between the service provider and user.
SLi	Substrate Link - A directed point-to-point link in a SN.
SN	Substrate Network - A graph that represents the nodes and links of the physical network with their resources.
SNo	Substrate Node - A node in a SN with a specific location/domain and a specific type (server, network device, or application device).

TAS	Time-aware Shaper - A traffic shaper from the TSN standard that provides deterministic latency using traffic scheduling.
TCT	Time-Critical Traffic - The traffic flows with strict timing requirements.
TSN	Time-Sensitive Networks - A recent IEEE standard for industrial Ethernet.
VLi	Virtual Link - A directed point-to-point link in a VN.
VM	Virtual Machine - An operating system hosted on top of another operating system using a hypervisor.
VN	Virtual Network - A graph that represents the nodes and links requested by the user with their requirements.
VNE	Virtual Network Embedding - Algorithms for allocating VNs on SNs.
VNF	Virtual Network Function - A network service executed in a virtualized form using a VM or a container running on a standard server.
VNo	Virtual Node - A node in a VN that represents the demand for computational capacity, certain capability of the host, or a specific type of phyical devices in a specific location/domain.
VNR	Virtual Network Request - The user-defined request for deploying a VN including VNos and VLis with their demands.

1 INTRODUCTION

Applying network virtualization technologies to complex and heterogeneous environments promises to reduce the Operational Expenditure (OPEX) and Capital Expenditure (CAPEX) and satisfy the increasing requirements. However, there are different virtualization techniques and different application levels in which various objectives might be of higher importance. Furthermore, network virtualization imposes challenges such as performance degradation; wider attack surface; considering multiple objectives and constraints that might be conflicting [1][♮1]; considering multiple network levels; autonomic composition of environment-aware virtual networks; efficient deployment algorithms; applicability to complex environments; and integration with real, mainly modern, technologies. These challenges are mostly considered partially and separately by researchers with a limited view on applicability.

1.1 NETWORK VIRTUALIZATION

Virtual Network Embedding (VNE) is a graph-theory-based domain that develops abstraction models of Virtual Networks (VNs) and their requirements and Substrate Networks (SNs) and their resources. Furthermore, VNE develops graph algorithms to map the VN requirements on physical resources. VNE algorithms calculate paths and allocations of server and network resources to a VN composed of Virtual Nodes (VNos) and Virtual Links (VLis) connecting them. The resulting calculations shall be used by the network controller to realize the actual resource reservation in the SN.

In another dimension, Network Function Virtualization (NFV) is a modern network virtualization technology that decouples the network functions from the proprietary hardware. These functions are executed in the form of Virtual Network Functions (VNFs) running on standard servers and chained in Service Function Chains (SFCs) to form end-to-end communication services. The SFC might include one or more sub-SFCs that determine the data flow paths based on traffic specifications. NFV leverages virtualization technologies to flexibly deploy complex network functions on-demand in the required locations. Multi-objective optimization is widely addressed by researches of network virtualization but they target the significant constraints partially and propose heuristics for large scale problems.

1.2 INDUSTRIAL ENTERPRISE

The traditional legacy networking paradigm of deploying proprietary and hardware-based network functions is not flexible enough to satisfy the emerging and ever-growing application requirements, mainly the security, performance, and resilience requirements of 5G mobile networks and Industry 4.0. This fact is due to the complexity and high cost of deploying new devices, upgrading devices, and service innovation.

1 The citations marked with ♮ refer to publications authored or co-authored by the author of this thesis.

Industry 4.0 enterprise applications will support the smart factory through data analysis and autonomic decision making. For this purpose, monitoring, analysis, and management functions at different levels of the enterprise are needed. For example, remote asset management is required when the administrators manage different distant locations to remotely control customizable production, safety, energy consumption, utilization, and security. IIoT enables remote asset monitoring by installing wireless sensors throughout the factory and integrating them via an Internet gateway. The data can be remotely accessed to take further actions in real-time.

However, in addition to the application functions in such a scenario, the service provider shall be able to create and deploy communication services that can satisfy the specific requirements of the industrial enterprise related to the monitoring traffic and resultant decisions. Virtualization technologies support the flexibility of the management applications by deploying data acquisition and analysis functions at the edge and central data centers, respectively. In this scenario, the central data center hosts enterprise functions, while the edge data center hosts individual factory functions.

The target applicability domain of this thesis is future dense industrial networks where multi-objective optimization methods are not efficient. We provide the base design of a complete virtualization system of a smart and complex enterprise. In this system, we target the main significant objectives, propose novel heuristics, and adapt some existing appropriate algorithms for the applicability domain.

1.3 SOLUTION APPROACH

In this thesis, we address the challenges of network virtualization with multiple combined virtualization approaches. We first research the comprehensive and autonomic network virtualization for complex environments that combines different techniques, multiple application levels, and the main objectives/constraints: topology, security, performance, reliability, and resource utilization. We focus on two main network virtualization approaches (dimensions); NFV for functionality virtualization and VNE for resource virtualization. We apply these dimensions to different levels of industrial networks under multiple objectives. We first propose a combined solution, then we apply certain technologies to certain levels under the most critical objectives/constraints. Furthermore, we design novel virtualization solutions and algorithms that provide high efficiency and usability. These solutions are supported by a specific use case for future industrial enterprise and integrated with two main components from the Time Sensitive Networks (TSN) standard.

We first develop a comprehensive VNE-NFV model for deploying an Extended Virtual Network (EVN) on a set of data centers. We introduce the concept of EVN that combines the virtual application and network functions, and the application and network devices. This EVN is built by applying a stepwise graph transformation on a simple VN (Application Request (AR)) defined by the user to represent the application end-nodes and general requirements. A set of policies (e.g., security policies) is also defined by the user and processed with a particular priority to extend the AR with specific VNos, VLis, and VNFs with the respective demands for resources. The policy represents a certain pattern in the EVN and a specific graph transformation operation to be performed when this pattern matches the current state of the EVN. The network model and policies consider the locations/domains and types of nodes (topology), and security, redundancy (reliability),

low latency and load balancing (performance) requirements. However, the cost reduction (resource usage) objective is considered in the embedding stage. This objective directly influences the energy consumption reduction objective. An example of a transformation is adding specific security VNFs when the VLi connects two locations.

The following step to building the EVN is composing the candidate SFCs based on the required VNFs and dependencies among them and using the topological sorting method. The embedding algorithm first allocates the virtual application nodes, and then it maps the SFCs using a greedy heuristic that adapts to the residual SFC and path lengths. This heuristic is designed to be feasible for our use case, and this also applies to a random topology used for evaluating it. The solutions are implemented in the VNE tool ALgorithms for Embedding of VIrtual Networks (ALEVIN) [11], an open-source VNE framework written in JAVA. A typical use case, remote asset management, from the IIoT domain is leveraged to illustrate the developed methods. Remote asset management is an enterprise application but is deployed over three levels: factory hall, edge computing, and cloud computing. However, in the general virtualization model, the factory hall and edge computing are merged to represent the factory level.

The comprehensive VNE-NFV solution that considers all objectives depends on three sub-solutions that consider performance, reliability, and security objectives for different levels and virtualization dimensions. For performance and reliability, we investigate the applicability of NFV and VNE, respectively, to a modern industrial networks technology (TSN). The level of applicability here is the factory in the sub-solutions and the enterprise in the combined solution. We apply NFV to a traffic shaping component (Time-aware Shaper (TAS)) to increase the flexibility, and we investigate the performance overhead as the main challenge with this approach. We also apply VNE to the redundancy component (Frame Replication and Elimination for Reliability (FRER)) to increase the flexibility while minimizing the resource usage as a significant challenge with reliability. For the security perspective and sub-solution, we apply both NFV and VNE at the enterprise level by proposing approaches for autonomic security management in the Network Function Virtualization Infrastructure (NFVI). NFV is applied via VNF migration-based defense architecture, and VNE is applied via a mapping algorithm that is aware of the security requirements of the VN and the security capabilities of the SN.

From the performance perspective, we study an essential traffic shaper from the TSN standard, TAS, and its existing implementations. We then research the mechanisms that can be used to implement virtual TAS using high-performance NFV. Based on these studies, we design, implement, and evaluate a preliminary virtual TAS as an SFC composed of multiple TAS-capable VNFs built using the Data Plane Development Kit (DPDK). Furthermore, we develop a complete framework with NFV-specific TAS controller, scheduling and transmission selection algorithms, time synchronization, and traffic generation and performance measurement tools. We propose a new method based on network prefetching to support the schedule calculation, mainly in virtual environments, by measuring the real transmission and processing times in advance. In the evaluation, we measure the frame loss and delay of Time-critical Traffic (TCT) traversing a TAS SFC using different scenarios to judge the capability of virtual TAS in providing comparable performance to the hardware-based TAS designed in the standard. Additionally, we evaluate the effect of Best Effort Traffic (BET) traversing the same SFC, and external disturbance traversing another SFC that uses the same resources. These factors are significant in analyzing the virtualization overhead.

From the reliability perspective, we adapt a model of the reliability as a property of network entities and develop reliability-ware and branching-based link mapping algorithms with different path disjointness policies that can be used with different industrial traffic classes. These algorithms depend on the traditional block diagram method for calculating path reliability. The algorithms are compared mainly in terms of resource utilization, VN admission ratio, and achieved reliability. A typical evaluation topology from industrial networks that includes various application types is used. Furthermore, we analyze the TSN sub-standard IEEE Std 802.1CB FRER that defines methods for stream replication. We build on it to develop compatible VNE models and use the VNE tool ALEVIN to export the mapping results in a format compatible with FRER. We then test this integration between the branching algorithms and FRER in the TSN simulator Tsimnet [70]. We developed the minimal and maximal branching methods and applied and evaluated them in industrial environments since these are efficient heuristics for complex environments that can improve the reliability with minimal or reasonable resource usage.

From the security perspective, we elaborate on the VNF security. The enterprise VNFs shall be deployed on standard servers that might expose a broader attack surface. We analyze the main security threats on the VNFs originating from the co-hosted VNFs. The threats under focus are side-channel, co-location, and migration exploitation attacks. We then design a defense concept based on a decision engine and VNF migration. The decision engine migrates the suspected VNF as a source of threat to a detailed analysis environment. Furthermore, we develop a security-aware node and link mapping algorithm that considers the security constraints of the VN and the security functions of the SN. In this thesis, we discuss Service Level Agreement (SLA)-related VNF placement policies only in the security and privacy context and relevant QoS issues.

1.4 CONTRIBUTIONS

In summary, we develop, adapt, apply, and evaluate several efficient heuristic algorithms for complex environments in which the optimization problems are inefficient. In the performance and reliability perspectives, we focus on the applicability of virtualization to industrial networks technologies. In the following, we list the contributions of this thesis to the state of the art:

- We propose a virtualization model that considers network levels, virtualization dimensions, and main objectives/constraints. Furthermore, we map the developed models and algorithms to this model. Figure 1.1 shows the proposed virtualization model with two network levels, both virtualization dimensions, and target objectives. The mapping of the developed approaches (combined and sub-approaches) is discussed in Chapter 7. The order of objectives here reflects priorities discussed in the combined approach in Chapter 3.

- We develop a comprehensive solution for combining virtualization techniques and applying them over different network levels and with multiple objectives.
  - We use graph transformation to convert the general application requirements to a physical-topology-aware EVN under different topology, performance, reliability, and security objectives and constraints. However, the resource utilization objective is addressed by the EVN mapping stage.

  - We develop and implement a greedy heuristic algorithm to embed an SFC depending on the chain and path length.
  - We apply the combined solution to an IIoT use case inspired by Industry 4.0 concepts and represents multiple levels and objectives.

- We develop a VNE model for reliability and three reliability-aware link mapping algorithms (shortest path, maximal branching, and minimal branching) and compare them in terms of VN acceptance, resource utilization, runtime, and achieved reliability. We use a typical factory topology with different applications for the evaluation. Based on our findings, we propose a mapping between traffic classes from industrial networks and these algorithms.

- We integrate theoretical network virtualization approaches with an industrial network technology (TSN).
  - We design and implement a complete framework for virtualization TAS using a high-performance NFV technique. Our solution achieves high flexibility in realizing TSN with accepted delay probability for enterprise-level industrial applications.
  - We integrate the mapping results of the reliability-aware link mapping algorithms with the standard IEEE Std 802.1CB FRER using the TSN simulator Tsimnet.

- We discuss VNF security threats and design a defense solution in NFVI based on migration, which considers relevant privacy and QoS aspects.

- We develop a security-aware VNE algorithm that maps the security constraints of the VN to the security capabilities of the SN.

1.5 THESIS STRUCTURE

This thesis is structured as follows: in Chapter 2, we introduce the work related to the virtualization of industrial networks, our combined approach, and our sub-solutions for performance (virtual TSN), reliability (branching and FRER), and security (migration-based defense and security-aware VNE). In Chapter 3, we present the combined EVN approach. In Chapter 4, we present the performance sub-solution of virtualizing TAS. In Chapter 5, we present the reliability sub-solution of branching and integration with FRER. In Chapter 6, we present the security perspective with the decision engine and security-aware mapping algorithm. Chapter 7 concludes this thesis by mainly mapping the developed methods to the virtualization model.

Figure 1.2 shows the thesis map with the logical elements and main dependencies. The contributions are mapped to the proposed virtualization model in the conclusions (Chapter 7) after detailing them in the respective chapters. The transformation logic in Chapter 3 is realized by different transformations that reflect the target objectives. The latency, redundancy, and security transformations and the EVN embedding algorithm use algorithms and concepts from the three sub-approaches in Chapters 4, 5, and 6, respectively. Finally, the related work aspects (EVN and the three main objectives) are connected to the respective chapters and clarify the state of the art and our contributions.

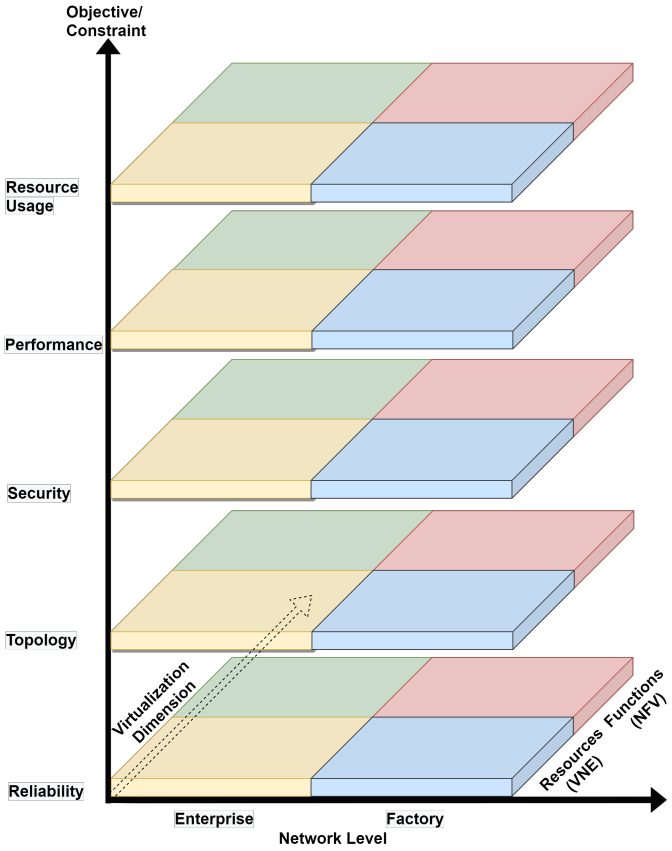

Figure 1.1: Proposed virtualization model

The applicability aspect in Chapter 2 (virtualization of industrial networks) is relevant to all approaches. However, this dependency is not shown for simplifying the figure.

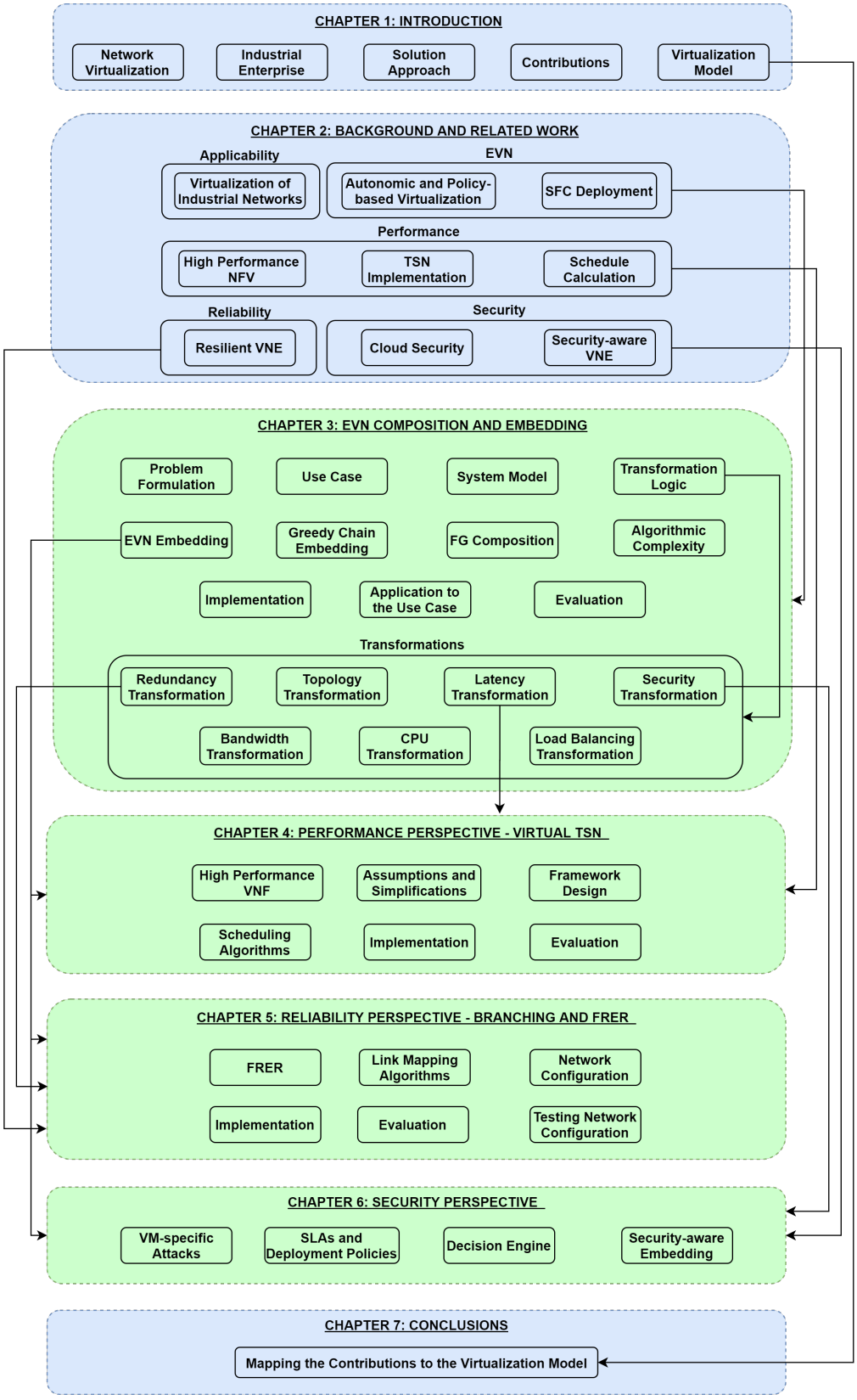

Figure 1.2: Thesis logical elements and their main dependencies

2 BACKGROUND AND RELATED WORK

This thesis includes multiple approaches for which different related research directions exist in the community. Since the EVN approach integrates these approaches, we combine all addressed related work in this chapter to provide an overview of the state of the art in the domains of network virtualization and its applicability to industrial networks. Furthermore, we highlight our contributions by comparing our approaches. However, due to the wide domain of relevant topics, we do not provide a complete survey rather an overview of the significant solutions that are relevant to our work with the main references.

2.1 VIRTUALIZATION OF INDUSTRIAL NETWORKS

The research on the virtualization of industrial networks mostly focuses on NFV. Some researchers investigated the applicability of NFV to industrial environments. Sakic et al. [13] proposed VirtuWind, an SDN and NFV architecture for industrial networks applied to wind parks with the focus on security VNFs and Supervisory Control and Data Acquisition (SCADA) as the main monitoring and management component deployed locally on-site. The architecture addresses a multi-operator ecosystem with inter-domain and intra-domain communication.

Although we discuss the related work on high performance NFV in Section 2.3.1, we discuss here its applicability to industrial networks. Few researchers investigated the capabilities of virtualization technologies for hosting real-time applications. Gundall et al. [14][1] examined the possibilities of virtualization in the industrial landscape. Performance comparisons for bare-metal applications, Virtual Machines (VMs), and containers were conducted and evaluated for industrial needs. The analyses indicate that the flexibility gained by virtualization can be achieved for industrial applications without violating the stringent real-time requirements in the industrial landscape, if Docker containers with a suitable configuration are used. Linux real-time kernel combined with Docker can reliably run the cyclic execution of applications in intervals of $1\mu s - 1ms$ with a jitter of $15\mu s$. The authors compare different Docker network drivers but do not develop a traffic shaper. From a security perspective, the authors recommend traffic encryption, but conclude that a non-negligible performance penalty does not allow network encryption for time-critical applications, then security must be achieved through a high degree of traffic isolation and other security measures. The authors suggest virtualizing the industrial control system at the edge and cloud level, which opens a vast surface of attacks.

Moga et al. [15] studied the capabilities of containers to achieve flexible consolidation and easy migration of industrial automation applications, as well as the container technology readiness with respect to the fundamental requirement of industrial automation systems, namely performing timely control actions based on real-time data. The authors provide an empirical study of the performance overhead imposed by containers based

1 This work is published after the author's publications [2]♮and [3]♮.

on micro-benchmarks that capture the characteristics of targeted industrial automation applications.

Bag et al. [16] explored the capabilities of different virtualization platforms offered by various cloud providers. The platform performance has been evaluated by conducting experiments with an industrial application, focusing on typical industrial aspects such as latency, jitter and availability. The findings indicate that specific application requirements affect the performance. Hence, application-specific evaluations might be necessary before deciding where an industrial application can be deployed.

The integration of cloud computing with industrial networks has been also addressed by Asenjo et al. [17]. The authors introduced a cloud-based virtualization generation service that collects data from multiple industrial automation systems of various industrial customers for storage and analysis on a cloud platform. A virtualization management component analyzes the data and generates a virtualized industrial automation system based on the analysis results.

In another relevant aspect, several works focused on sensor and gateway virtualization in wireless sensor networks in the IoT domain. Mouradian et al. [18] proposed an NFV architecture for virtualized wireless sensor and actuator network gateways, in which software instances of gateway modules are hosted in NFV infrastructure. Furthermore, some works focused on virtualizing the industrial network control. For example, Lee et al. [19] virtualized CAN controllers using containers.

In the VNE domain, Huth and Houyou [20] proposed a system architecture for network virtualization in industrial networks. A domain controller, the Slice Manager, directly manipulates a potentially heterogeneous network while providing a simple abstract view to planning and management applications. The application runs in a "slice" which is a VN with clear QoS guarantees and bandwidth policies.

In summary, the research on the virtualization of industrial networks does not combine VNE and NFV and all important constrains, and does not autonomically build VNs that represent different devices, gateways, VNFs, and virtual application functions. Furthermore, TSN and virtualizing traffic shapers in the form of SFCs are not addressed. Compared to the above-mentioned works, we consider edge and cloud computing for data analysis with inter- and intra-domain and location communication, we combine traditional VNE and NFV, and target the objectives of security, performance, reliability, and resource usage. We compose an EVN that might include all types of devices, gateways, different types of VNFs, and virtual application functions. For security considerations, we use a VM-based approach in our virtual TAS since VMs offer higher level of isolation and lower performance, but reasonable for enterprise applications. In the reliability aspect, we use the concept of network slicing in industrial networks with different applications to compare different redundancy methods.

2.2 COMBINED APPROACH

The combined EVN approach represents an autonomic, policy-based, and multi-objective network virtualization using traditional VNE algorithms and SFC composition and deployment solutions. In this section, we address the related work on these topics.

2.2.1 *Autonomic and Policy-based Network Virtualization*

Several works addressed the autonomic virtual networking that creates the VNs from policies, with the focus on multiple but incomplete objectives. Davy et al. [21] presented an approach to provision network services in an autonomic network using virtualized routers. The approach provides business users a method of describing the requirements and behavior of a set of network services using policies, while abstracting the users from complicated network configuration tasks. It then dimensions a VN dedicated to provisioning these services. The authors focus on fault tolerance, redundancy, and security.

In a later work, Davy et al. [22] addressed a specific scenario of secure VPN services that require a set of security-related functionality from the network to be effectively deployed. In a similar direction, Louati et al. [23] designed an architecture of autonomic virtual routers to support automated provisioning and management of VNs. The objective is to automatically create virtual routers in Substrate Nodes (SNos) to support on-demand VNs, without any human intervention.

Granelli et al. [24] proposed an architecture to realize autonomic mobile VN operators that can be deployed by Internet service providers to guarantee efficient and effective network adaptation to unexpected events and real-time resource requests. Mijumbi et al. [25] proposed to use distributed artificial intelligence to make the VNs self-configuring, self-optimizing, self-healing, and context-aware.

In comparison, we focus on both autonomic VN composition and configuration. Furthermore, we consider comprehensive policies with certain priorities and different types of VNFs and combine the traditional VN with SFCs. Our work focuses on industrial enterprises with multiple levels and on TSN technology. Our composition stage adds physical devices automatically and considers locations and domains. However, we don't target dynamic or proactive autonomic virtual networking that reconfigures the VNs on-demand or in advance based on learning methods as discussed by Yan et al. [26] and Rkhami et al. [27], who use graph neural networks.

2.2.2 *Multi-Objective VNE*

The research on multi-objective VNE focuses on optimization methods, mainly integer linear programming, and on developing heuristics for large scale environments. Furthermore, several works consider the embedding over multiple independent domains with the focus on limiting the domain's information disclosure. Table 2.1 summarizes the main existing solutions. However, solutions that focus on resilient VNE with other secondary objectives, mainly resource utilization, are discussed in Section 2.4. Additionally, the multi-objective and resilient SFC deployment are discussed in Section 2.2.3.

Our work considers several and comprehensive objectives and focuses on future dense industrial environments, making optimization problems inefficient. We combine VNE and NFV and consider physical devices that exchange data and that cannot be virtualized. Furthermore, we develop a set of heuristic algorithms to overcome the complexity of multi-objective optimization. However, compared to works that solve the multi-domain embedding problem with limited information disclosure, we do not consider domain privacy inside an industrial enterprise, and we consider both domains and locations.

Table 2.1: Important existing works on multi-objective VNE

Reference	Methodologies	Main objectives/constraints
Houidi et al. [28]	Mixed integer programming	Power consumption, availability, load balancing, fault-tolerance
Zhang et al. [29]	Artificial immune system	Revenues, energy consumption
Li et al. [30]	Mathematical programming	Acceptance ratio, resource load, profit, delay, load balancing
Gong et al. [31]	Compatibility graph	Location
Habibi et al. [32]	Graph neural networks	Scalability
Chowdhury et al. [33]	Heuristic	Multi-domain, location
Dietrich et al. [34]	Request partitioning	Multi-domain with limited information disclosure
Ni et al. [35]	Particle swarm optimization	Multi-domain, cost
Andreoletti et al. [36]	Reinforcement learning	Multi-domain with limited information disclosure
Zhang et al. [37]	Particle swarm optimization, heuristic	Multi-domain Internet of Drones, delay, cost
Yu et al. [38]	Heuristics: VLi splitting, path migration, VN topology classes	Resource utilization, delay
Fajjari et al. [39]	Greedy algorithm for VN reconfigurations to minimize the number of bottlenecked SLis	Admission, cost

2.2.3 *SFC Deployment*

We focus in this section on SFC composition and embedding. Herrera and Botero [40] provided an extensive survey on NFV resource allocation. They highlight the fact that most of the contributions focus on chain embedding and use linear programming to optimize the solution for a certain objective, such as runtime, end-to-end latency, and deployment cost. However, several researchers studied the SFC composition problem.

Mehraghdam et al. [41] introduced a context-free language model to formalize the VNF requests and a heuristic to compose the SFCs. SFC allocation is formulated as an optimization problem with different objectives. The authors highlight the fact that placing SFCs is complex, in particular for different, possibly conflicting, allocation objectives, such as used number of SNos or latency. An optimal SFC composition approach based on integer linear programming is introduced by Ocampo et al. [42]. The objective is to find SFCs with minimal bandwidth requirements for a predefined VNF request. Gil-Herrera and Botero [43] proposed a meta-heuristic algorithm for solving the SFC composition stage. The solution focuses on minimizing the total bandwidth demand.

Zheng et al. [44] studied how to optimize the latency in hybrid SFC composition and Embedding. They proposed an approximation algorithm (Eulerian Circuit-based) to jointly optimize the SFC construction and embedding. Furthermore, the authors propose an efficient Betweenness Centrality based algorithm. Wang et al. [45] formulated the SFC composition and mapping problem as a weighted graph matching problem with the focus on resource optimization. The authors also proposed a Hungarian-based algorithm to solve the SFC composition and mapping problem in a coordinated way.

Beck and Botero [46] introduced a heuristic coordinated approach, CoordVNF, which tackles the chain composition and embedding problems with the objective of minimizing the bandwidth usage. The approach uses backtracking in case of an invalid embedding to find a valid solution starting from the last successfully embedded VNF. Hirwe and Kataoka [47] proposed an approach, referred to as LightChain, which creates a directed acyclic graph for a VNF request and performs topological sorting to generate a single SFC. This SFC is then embedded on the shortest path between source and destination devices based on the assumption that a path of length n can host n VNFs. If the current path is consumed, another shortest path is calculated for placing the remaining VNFs.

Wang et al. [48] classified the SFC deployment problems according to the option of sharing VNF instances. An optimization problem and a heuristic solution are presented to find the best SFC composition scheme that achieves minimal demand of link resources and improves the VNF instance utilization. The stages of SFC composition, placement, and assignment are combined. Li et al. [49] presented a solution for placing VNFs in cloud data centers. They focused on the placement of multi-tenant VNFs shared among multiple SFCs to achieve efficient utilization of network resources in contrast to traditional VNF placement strategies.

Wang et al. [50] introduced an SFC composition framework called Automatic Composition Toolkit. It tries to automatically detect the dependencies and conflicts among the network functions and model their behavior. The authors define topology and processing dependencies, and action and processing conflicts. Li et al. [51] presented a typical three-stage coordinated optimization model for NFV resource allocation, which considers CAPEX, OPEX, and link costs.

From another perspective, some researchers addressed the problem of resilient allocation of SFCs. Wang et al. [52] introduced a model for calculating the SFC availability and presented a Joint Path-VNF backup model that jointly considers path and VNF backup. Furthermore, they used a priority-based algorithm to optimize the composition and mapping of SFCs. In this algorithm, VNF dependency is converted to a VNF priority, which might create multiple Forwarding Graphs (FGs) generated according to these priorities. The evaluation metrics are total data rate, acceptance ratio, maximum availability, VNF cost, and physical node and link cost.

Beck et al. [53] discussed two main resilience strategies: VLi and VNF resilience. The VLi resilience strategy tries to calculate a disjoint backup path when mapping a target VLi that connects two VNFs. VNF resilience needs at least two different SNos on which the VNF is allocated. The evaluation focused on the acceptance ratio and embedding cost. Torkzaban and Baras [54] integrated the trustworthiness of the SN paths into the SFC embedding problem. The authors formulated the path-based trust-aware SFC embedding problem as a mixed integer-linear program, and provided an approximate model based on selecting the shortest candidate SN paths.

Regarding the consideration of domains and locations, Liu et al. [55] proposed a distributed method for cross-domain SFC embedding. Besides preserving the privacy and autonomy of domains, the authors consider fair competition and migration-based load balancing among domains, and improving the admission ratio. The SFC is partitioned and mapped in each domain according to its specific policies. Lange et al. [56] proposed a multi-objective heuristic for the optimization of SFC placement. The objective is to determine the number, location, and assignment of VNF instances and the routing of demands. The constraints of CPU utilization and the delay of individual flows are taken into account.

Our EVN solution contributes to the current state of the art of the three topics of this section in different aspects. First, a policy-based transformation of an abstract AR composes an EVN that includes the physical and virtual application nodes and the VNF requests. Our policies consider primary topological, performance, security, and reliability requirements. Second, we integrate location- and domain-awareness into the EVN composition and embedding solutions. These solutions consider dependencies among VNFs, which also reflect the locations and domains. The transformations add security VNFs according to the domains and locations across which the traffic flows. Third, we define a specific use case of these approaches from Industry 4.0 and design the fixed topology that represents it. The last contribution is an SFC embedding heuristic that tries to find a valid solution for the embedding problem while using a pre-verification of the path. To the best of our knowledge, this is the first work that addresses the composition of an EVN that combines the application and SFCs, based on multiple policies, while considering domains and locations, and in the context of industrial environments.

2.3 PERFORMANCE PERSPECTIVE - VIRTUAL TSN

Our performance solution virtualizes a TSN component using a high performance NFV mechanism. Furthermore, we present a novel adaptive scheduling method for virtual environments. The important related work on these topics is discussed in this section.

2.3.1 *High Performance NFV*

Several works and tools addressed high performance NFV. Intel presented DPDK [57]; a set of performance-boosting libraries for NFV architecture. Sun et al. [58] presented HYPER, a framework that integrates hardware and software infrastructures to provide flexibility while keeping high performance. Naik et al. [59] introduced libVNF as a C++ library that uses DPDK, among other technologies, to ease the implementation of horizontally scalable high performing VNFs. The developers of libVNF claim reducing the source code for developing a VNF by 50% with a maximum decrease of performance of 10%. Nakajima et al. [60] proposed a high-performance virtual NIC framework for hypervisor-based NFV with user space virtual switch and DPDK, with the focus on improving the virtual NIC throughput.

Kourtis et al. [61] employed deep packet inspection to evaluate the performance of Single-root Input/Output Virtualization (SR-IOV) with DPDK- based NFV deployment. Li [62] built a high performing software router based on DPDK to analyze the IPv4

header of a packet and determine whether to forward it or submit it to the upper layer protocol. The author compared his work to the Linux IP protocol stack and found that his implementation increases the throughput by 8-10 times. Yurchenko et al. [63] presented OpenNetVM as an open-source NFV platform designed to accelerate the development of VNFs by creating an abstraction layer over DPDK. OpenNetVM allows Docker container-based VNFs to be chained together and run on the same host with minimal latency.

Yasukata et al. [64] introduced HyperNF, a high performance NFV framework to increase server throughput when concurrently running large numbers of VNFs. HyperNF implements hypercall-based virtual I/O, placing packet forwarding logic inside the hypervisor to significantly reduce I/O synchronization overheads.

Some works used different hardware technologies to achieve high performance NFV. Sun et al. [65] integrated a stateful and NetFPGA accelerated data plane into NFV. The authors designed a performance-aware service chaining algorithm to fulfill both functionality and performance requirements with respect to SLAs. They implemented an SLA-NFV prototype based on OpenStack and NetFPGA. Zheng et al. [66] proposed a GPU-based high performance framework for NFV. The elasticity of VNFs is improved by scaling them up and down by allocating a different number of fine-grained GPU threads to a VNF during runtime.

In comparison to these works, we use DPDK-accelerated Open vSwitch and VM-based VNFs. We focus on delay and frame loss and develop a specific shaper from TSN with a complete framework that includes schedule calculating and distribution methods.

2.3.2 *TSN Implementations and Integration*

In order to evaluate the networks implementing TSN features, few simulators have been developed. TSN capabilities of timed transmission were addressed by Jiang et al. [67], who presented a TSN simulation model built on top of OMNET++ [68]. TAS-enabled switches were simulated through the calculation of Gate Control Lists (GCLs) with the opening times of the TCT gates. The authors tried to analyze the end-to-end latency of scheduled and non-scheduled traffic. Jarray et al. [69] presented NeSTiNg, which implements the frame tagging according to 802.1Q; a TSN component whose functionalities include forwarding, queuing, and two kinds of shapers (TAS, and credit-based shaper).

Heise et al. [70] presented TSimNet (on top of the Inet [71] framework) that we use in our work with FRER. The authors implemented the non-time-based features of TSN such as frame preemption, per-stream ingress policing, and FRER. The simulator was used for purposes of evaluation in avionic networks. Pahlevan [72] presented another simulator that implements the time-based and non-time-based features of TSN (time-based filtering, TAS, policing, and FRER), based on Riverbed simulation framework. Lee and Park [73] developed a simulator for in-vehicle networks and designed a model for autonomous vehicles with TSN features.

Some researchers evaluated the features of TSN to understand the advantages of using them. Pahlevan and Obermaisser [74] evaluated the safety and fault tolerance offered by FRER in Opnet simulation framework. The authors verified that FRER is resilient in case of transient errors (e.g., stuck transmitter) and in case of permanent errors (e.g., link failure). The applicability of the TSN standard has been also evaluated by Pahlevan and

Obermaisser [75], where the Opnet simulation framework has been used to compare the obtained results to the theoretical performance defined by the standard.

Hardware with integrated TSN features also exists. An Intel Platform Designer component [76] has been developed to make it easy to integrate TSN features in Intel Field Programmable Gate Arrays (FPGAs). The tool can be used for time synchronization and TCT scheduling with eight priority queues per port. Other devices from different manufacturers were developed to be fully (Broadcom [77]) or partially (Cisco [78]) compatible with TSN features presented by the time of their development.

The integration between TSN and other technologies has been addressed by several researchers. 5G-TSN integration for smart factories is currently under focus by some researchers such as Gundall et al. [79] and Farkas et al. [80]. According to these researchers, this integration is promising to fulfill the requirements of Industry 4.0 due to the flexibility features of 5G and the TSN feature of extremely low latency. However, these works do not address the real virtualizing of TSN using NFV, which is also a main enabler of 5G.

Pop et al. [81] discussed using TSN as a deterministic transport mechanism for fog computing in industrial automation. The authors proposed a configuration agent architecture based on IEEE 802.1Qcc and Open Platform Communications Unified Architecture (OPC UA). This architecture is capable of performing runtime network configuration to address the configuration challenges for scheduled networks. The authors also proposed a list scheduling-based heuristic to reconfigure the scheduled network at runtime for industrial applications within the fog. Li et al. [82] also integrated TSN and OPC UA for dynamic configuration. SDN was also used by Said et al. [83] as a solution for the dynamic configuration of TSN, and by Boehm et al. [84] with a combined control plane for SDN and TSN.

Regarding TSN virtualization, Fang and Obermaisser [85] implemented a TAS-capable virtual switch as a kernel module. In comparison, we deploy TAS-capable and VM-based SFC, use an adaptive GCL calculation method (prefetching), inject random BET and a disturbance SFC, consider frame loss in the evaluation, consider different evaluation scenarios from industrial enterprises, and build a complete framework with synchronization, schedule distribution, and performance measurement.

2.3.3 *Schedule Calculation*

A TSN performance aspect that is still under intensive research is the TAS schedule calculation. Some researchers discussed the best methods to calculate the GCLs for a network of bridges hosting multiple TCT flows. The goal is to reserve time slots for flows along their paths such that: the delay constraints of flows are respected, no overlap in the transmission of any pair of flows, and the total length of the schedule is minimized. Craciunas et al. [86] used Satisfiability Modulo Theories (SMT), a decision-making methodology, to calculate the schedule and assignment of flows to the existing queues on the ports. The flows are periodic with pre-defined cycles and frame sizes. The flow scheduling constraints are end-to-end delay, transmitting the frame inside the flow cycle, no overlapping of frames, and preserving the order of frames in the flow. Additionally, the authors introduced the flow isolation constraint; a queue is reserved for a flow until all its frames in the queue are transmitted. However, the SMT method is not scalable, and the authors used an incremental backtracking algorithm that schedules one flow

in each step using SMT. The results for each flow are considered when the next flow is processed. In case no scheduling for one flow is found, a step backward is made, and the last scheduled flow is mapped with the current flow in one step.

As an improvement to [86], Farzaneh et al. [87] proposed an approach to track the unsatisfiable flows and correct the schedule. Gavriluţ [88] proposed a Greedy Randomized Adaptive Search Procedure (GRASP) to calculate the GCLs. Their meta-heuristic approach generates a set of feasible solutions iteratively, and a candidate is randomly chosen as an initial solution. The solutions are then enhanced to reach the locally optimal solutions using the Hill Climbing strategy. The resulting solutions are compared based on an objective function that reflects the difference between the worst-case end-to-end delay and the delay threshold of a flow. This approach considers both the TCT and Audio Video Bridging (AVB) traffic. The resulting schedules respect TCT constraints with an accepted increase in the AVB delay.

Pahlevan et al. [89] presented a heuristic algorithm that routes the flows and creates the GCLs in one phase. However, they only consider TCT, and it is treated as tasks that can be assigned to the available nodes. In each node, one queue is used for TCT, and the flow isolation method is adopted. Scheduling a flow respects the periods of the already scheduled flows on the same link. Dürr [90] adapted the No-Wait Job Scheduling Problem (NW-JSP) to the TAS scheduling problem. The NW-JSP problem deals with scheduling a set of jobs on a set of machines. A heuristics algorithm is used to minimize the schedule length. Barzegaran et al. [91] used constraint programming for quality-of-control-aware scheduling of communication in TSN-based fog computing platforms.

Hellmanns et al. [92] proposed a scalable scheduling model for converged networks supporting different traffic types. They introduced a procedure for schedule planning of isochronous traffic, which exploits the hierarchical structure of factory networks. The authors split the network into sub-networks and use a two-stage approach based on a heuristic and a tracing mechanism. Arestova et al. [93] discussed a hybrid genetic algorithm including chromosome representation for the routing and scheduling problems in TSN. Additionally, the authors introduced an approach to compress the resulting schedules. Serna Oliver et al. [94] discussed how the synthesis of communication schedules for GCLs defined in IEEE 802.1Qbv can be formalized as a system of constraints expressed via the first-order theory of arrays.

In this thesis, the goal is not improving theoretical schedule calculation approaches but implementing and evaluating an efficient method in a real and virtual environment. We design and compare two scheduling algorithms that use a new mechanism of measuring the actual transmission and processing times in advance. This mechanism is significant in controlling the fluctuations, mainly in virtual environments. Our algorithms consider TCT and BET, and each port in a node includes one queue per traffic class. The first algorithm is empirical and uses experimental network prefetching data to calculate the GCL, and the second algorithm is hybrid and additionally uses the flow's frame size and link bandwidth.

We evaluate the performance of DPDK-based VNF implementation under the time-aware functionality and scenarios of TSN. While most of the high performance NFV works (such as [61] and [62]) evaluate the implementations in terms of computational overhead and throughput, we focus on delay and frame loss. Furthermore, we develop a complete framework that can be used to evaluate the performance of TSN and its future improvements in a real environment, with low cost and high flexibility.

2.4 RELIABILITY PERSPECTIVE - RESILIENT VNE

The research on resilient VNE focuses on the efficient reservation of backup resources and single Substrate Link (SLi) failure. However, some works consider reconfiguration as a defense mechanism against attacks, reliability values, locations, domains with privacy, and improving the VN topology. A summary of some important existing solutions is shown in Table 2.2.

In comparison to these works, we consider reliability values, VLi reliability, locations, domains, resource usage, and extending the VN topology in one approach. Furthermore, we develop minimal [2] and maximal branching as efficient heuristics and apply, compare, and evaluate them in industrial networks. We discuss a mapping between industrial traffic classes and these redundancy policies. Finally, we use the mapping results to compute the required network configurations according to 802.1CB FRER.

2.5 SECURITY PERSPECTIVE

The security solutions presented in this thesis consider SLA-aware and migration-based suspicious VM isolation, and security-aware VNE. The important related work on these topics is discussed in this section.

2.5.1 *Cloud Security*

Serious vulnerability concerns have arisen from the VM co-residence architecture of the IoT cloud (Xing et al. [110]). Co-residence attacks enable an attacker to access and corrupt a user's sensitive data by co-locating his VM on the same physical server [110]. In this thesis, we assume that the VM is the preferred technology for hosting VNFs that process industrial data for security considerations. This issue is further discussed and applied mainly with virtual TSN in Chapter 4. We assume here that the research about VM security applies to VNF security. However, we extend our VM security considerations to the SFC.

There are several attack vectors, threats, and defense mechanisms in IaaS clouds, but we focus on attacks on VMs from co-hosted VMs as the most relevant for sensitive industrial data. We don't target the threat of compromising the VM monitor and the management VM. We assume that the probability of a malicious co-located VM is higher. An overview of such attacks and specific defense mechanisms is presented in Chapter 6. Since this chapter includes a general defense architecture against malicious VMs while considering the SLAs, we focus on related work on similar cloud defense mechanisms and security SLA management in cloud computing. However, these topics consider other types of attacks performed by malicious VMs, such as DDoS. The exact detection mechanisms of such attacks are out of the scope of this thesis, rather the behavior patterns and reasonable reactions according to the SLA.

Many researchers addressed the behavior study of both traditional malware and virtual environment-specific malware in cloud environments. For example, Dolgikh et al. [111] propose an efficient behavioral modeling scheme to detect suspicious processes in client

2 Liu et al. [12] presented a similar concept in 2019. However, we developed the minimal branching algorithm in cooperation with Siemens in 2018.

Table 2.2: Important existing solutions for resilient VNE

Reference	Methodologies	Main objectives/constraints
Jarray et al. [95]	Column Generation, protection cycle	Single link failure, minimizing the backup resources
Chen et al. [96]	Heuristic, restoration path selection	Cost-effective usage of network resources
Jiang et al. [97]	Bipartite graph matching, resource sharing among working and backup facilities	Location, single facility failure, minimizing the bandwidth consumption
Oliveira et al. [98]	Multiple substrate paths, capacity reallocation	DoS attack
Chai et al. [99]	Multi-objective optimization problem, single-objective subproblems, discrete particle swarm optimization	Malicious attacks in SDN, minimizing network load, maximizing embedding reliability
Rahman and Boutaba [100]	Heuristic, node migration, restoration	Single SLi failures
Guo et al. [101]	Heuristics, failure dependent protection, enhanced VN by backup facility nodes, resources sharing, binary quadratic programming, mixed integer linear programming	Single facility node failure, efficient resource usage
Liu et al. [12]	Availability model, integer linear programming, heuristic, k-shortest paths, backup paths	Minimizing resource cost
Liu et al. [102]	Backup VN topology, resource sharing	SLi failure
Ergenc et al. [103]	Mixed integer linear program, heuristics, dynamic function migration	Arbitrary node failures, attacks
Chen et al. [104]	Reliability-aware mapping, partial protection, mapping permutation to adjust the reliability	Reliability, resource usage
Wu et al. [105]	Heuristic, reliability-aware mapping and backup of nodes, fault recovery by backup switching	Reliability
Zhang et al. [106]	topology- and reliability-aware node ranking, shortest path	Reliability
Rahman et al. [107]	Backup paths with bandwidth rerouting	Single SLi failure, delay
Gomes et al. [108]	Spanning tree, redundant paths, shortest path with bandwidth weight	Overall VN reliability, SLi failure
Yu et al. [109]	VNo migration, disjoint backup path	SLi failure, resource usage

VMs by monitoring system calls. Marnerides et al. [112] describe how to detect the traditional Kelihos malware in VMs by monitoring the memory usage and number of processes. According to the authors, Kelihos malware causes a memory explosion for few seconds, which is not a normal behavior for traditional applications. Zhang et al. [113] use machine learning to deploy a lightweight mechanism in a VM to detect the behavior of L2-cache side-channel attacks performed by other VMs. Several works address the defense mechanisms against suspicious/malicious VMs and co-location attacks [3]. The main existing solutions are summarized in Table 2.3.

Several researchers addressed the security SLAs in IaaS clouds. For example, Kaaniche et al. [130] proposed an extension to an existing SLA description language, rSLA, to describe the security requirements and needed security mechanisms that allow the automatic management of the security SLAs. The proposed system dynamically inspects the SLA document and activates the needed security monitoring tools to collect the data. The data is evaluated against the objectives to execute enforcement and reporting actions. De Benedictis et al. [131] discussed how to define a per-service security SLA in the cloud. Zhou et al. [132] presented a privacy-based SLA violation detection model for cloud computing based on Markov decision process theory. This model can recognize and regulate the provider's actions based on the specific requirements of various users. Additionally, the model can evaluate the credibility of the provider by monitoring the actions that violate the user's privacy. Basile et al. [133] studied the integration of network and security policy management into an NFV framework by enabling and configuring security VNFs based on the user requirements. Ullah and Ahmed [134] proposed integrating security levels in SLAs and VM placement.

Compared to the above mentioned works on defense and security SLAs, our primary defense mechanism tries to migrate a suspicious VM to a dedicated analysis environment, while respecting the downtime allowed by the SLA and migration history. However, we define a set of SLA policies described with our own format, and respective deployment decisions and actions to be taken based on the environment conditions. Our policies are defined at the VN/VM-level and cover different QoS, privacy, and security concerns in IaaS clouds.

2.5.2 *Security-aware VNE*

The research on security-aware VNE focuses on defining and mapping security levels. Bays et al. [135] considered locations, resource usage, and three distinct confidentiality levels in an optimization problem. In a following work, Bays et al. [136] developed a heuristic with end-to-end and hop-to-hop cryptography while considering processing and bandwidth costs. Zhang et al. [137] proposed a security-aware VNE algorithm based on reinforcement learning. The authors defined security levels for VNos and SNos. The VNo can only be mapped to a SNo with equal or higher security level. The training phase is used to assign SNos a certain security level probability. Wang et al. [138] used security-level-based node filtering. Liu et al. [139] presented an optimization problem and a heuristic that considers security levels and resource usage.

Some solutions improved the traditional security level-based solutions. Beşiktaş et al. [140] mapped VNs of conflicting operators to different SN entities. Liu et al. [141] defined

3 Works that suggest migration-based isolation and detailed analysis of suspicious VMs are published after the author's publication [4][‡].

Table 2.3: Main defense mechanisms against suspicious VMs and co-location

Reference	Methods	Main objectives/constraints
Hou et al. [114]	Recognizing potentially risky VMs based on historical data, consolidating risky VMs into specialized servers, risk-ware VNE heuristic	Risk detection and isolation
Mohamed et al. [115]	Risk-based VM placement	Selecting the host that leads to minimum risk increase
Bardas et al. [116]	Moving target defense: automated renewal of the instances	Preventing co-location and side-channels
Han et al. [117]	Learning techniques to classify users, two-player security game	Increasing the cost of co-location
Han et al. [118]	Security game with probabilistic selection of different VM allocation policies	Minimizing co-location possibility, workload balance, power consumption
Miao et al. [119]	Security-aware VM placement: rules, allocation, and migration based on conflicting tenants	Minimizing co-residency, load balancing, power consumption
Agarwal and Binh Duong [120]	Previously co-located users first	Minimizing co-location
Feizollahibarough and Ashtiani [121]	Security-aware VM placement based on VM vulnerability and importance levels and server vulnerability and capacity	Reducing the risk of co-location
Azab and Eltoweissy [122]	Moving target defense by probabilistic random migrations	Minimizing the probability of co-residency and side-channels
Zhang et al. [123]	Moving target defense by periodic migration	Increasing co-location difficulty, optimal migration intervals
Anwar et al. [124]	Game theory and periodic migration	Increasing co-location cost
Atya et al. [125]	Experimental study on co-residency and side-channels in Amazon EC2, guidelines for migration-based defense	Co-location, side-channels, migration costs
Liu et al. [126]	Optimization problem	Minimizing co-residence of multiple users, power consumption, workload balancing
Bedi and Shiva [127]	Two-player game	Co-location with DoS
Deyannis et al. [128]	Offload the processing of malware analysis to a remote server	User data privacy, performance overhead
Casola et al. [129]	Moving target defense by automatically switching among different admissible application configurations	Security SLA

an optimization problem and two heuristics that consider physical isolation, security levels, splittable VLis, and resource usage. Wang et al. [142] discussed the security issues at nodes, topology, and network levels. The authors presented flexible and fine-granular security plans and used a path-based mathematical model and node filtering.

In comparison to these works, we consider locations and domains (in the EVN) and add the required security VNFs according to certain policies. However, our original security-aware VNE algorithm maps specific security demands to the respective security functions offered by the hosts and substrate domains. Furthermore, this algorithm forces Cross-domain Links (CDLs) to be mapped across firewalls.

2.6 CONCLUSION

Network virtualization solutions use optimization problems in addition to heuristics and approximation methods for efficiency. The solutions either consider only VNE or combine it with NFV, and consider some important objectives. However, there is no comprehensive and efficient solution for complex environments that considers all significant constraints. Furthermore, there is no solution for the autonomic creation of VNs that combine the network and the applications and consider different prioritized policies. Finally, the integration with industrial networks focuses on security VNFs, control, and analyzing the performance without virtualizing the communication technologies in the form of isolated VMs. This thesis addresses these challenges.

3

EVN COMPOSITION AND EMBEDDING

This chapter is an extension of the author's publication [3][‡].

The autonomic composition of VNs and SFCs based on application requirements is significant for complex environments. In this chapter, we use graph transformation in order to compose an environment-aware EVN based on comprehensive objectives and constraints. The EVN can represent physical devices and virtual application and network functions. We build a generic VNE framework for transforming an AR into an EVN. Subsequently, we define a set of transformations that reflect primary topological, performance, reliability, and security policies. The resource usage objective is considered by the EVN embedding stage and is directly related to energy consumption. Furthermore, we further discuss privacy-aware cloud security policies in Chapter 6. The transformations update the entities and demands of the VN and add SFCs that include the required VNFs.

Additionally, we propose a greedy heuristic for path-aware embedding of the composed SFCs. This heuristic is appropriate for real complex environments, such as industrial networks. Furthermore, we present an IIoT use case inspired by Industry 4.0 concepts. In this use case, EVNs for remote asset management are deployed over three levels; manufacturing halls and edge and cloud computing. We also implement the developed methods in ALEVIN and show exemplary mapping results from our use case. Finally, we evaluate the chain embedding heuristic using a random topology that is typical for such a use case, and show that it can improve the admission ratio and resource utilization with minimal overhead.

3.1 INTRODUCTION

The SFC composition problem has already been discussed by researchers without addressing different application- and service provider-driven policies. Such policies might be required, for example, to place certain VNFs in certain locations or to add certain security functions according to the traffic path and application security requirements. From another perspective, solving the SFC composition problem without considering the application functions imposes inconsistency between the composed SFCs and the application requirements. For example, when the application functions are also virtualized, some functions might need to be instantiated over multiple servers for fault tolerance, load balancing, or server capacity constraints. Such a case requires modifications to the composed SFC, for example, by adding a load balancing VNF between the SFC and instances of the application function.

The problem of embedding an SFC is similar to traditional VNE problems. In VNE, the physical network (SN) and the VN to be embedded (overlay with VNos and VLis) are represented as graphs. VNE algorithms apply optimization and heuristic methods to find the optimal mapping of the VN on the SN with different objective functions, such as admission ratio, embedding costs, path length, or delay. VNE algorithms are needed

in NFV to perform the autonomic composition and embedding of SFCs based on the application requirements and service provider policies.

In this chapter, we present a model that combines NFV with the node and link mapping approaches from VNE. This model transforms an abstract AR that defines certain properties of end-nodes, such as type and location/domain, and it might define primary security, low latency, and redundancy requirements. The AR is converted using the graph transformation method to an EVN that combines the application end-nodes (physical or virtual) with the composed SFCs based on the rules defined by the service provider. The graph transformation sequentially builds the EVN from the AR based on policies related to devices'/functions' types and locations/domains, redundancy, low latency, security-related network functions, and load balancing. The communication services among VNos are realized via VNF requests where needed.

Several FGs are possible based on the created VNF request and dependencies among VNFs. We present an approach for creating all possible FGs using topological sorting, and location/domain-based verification of the candidate FGs. The EVN embedding algorithm first embeds the application VNos and VLis that do not include VNF requests. Subsequently, the SFCs (FGs) are embedded using a greedy heuristic that depends on the chain and path lengths. The model is supported by a use case, remote asset management, inspired by the smart factory concepts in Industry 4.0 and based on IIoT concepts.

The motivation behind these methods is to enable autonomic virtual networking in complex environments with minimal user intervention and complexity in defining the application requirements. Furthermore, such a set of polices enables the service provider to satisfy several objectives with feasible priorities. A complete combined VNE–NFV framework that considers the topology and properties of the physical network and application requirements is needed for service provides to be able to flexibly and seamlessly deploy applications in a complex environment.

The rest of this chapter is organized, as follows; Section 3.2 describes our problem, use case, and system model. Section 3.3 details our VNE methodologies related to AR transformation and EVN and chain embedding, and discusses the algorithmic complexity of these methodologies. Section 3.5 presents our implementation in ALEVIN, the composition and mapping results for a representative AR from the use case, and the evaluation of the chain embedding algorithm with a random topology. Section 3.6 concludes this chapter.

3.2 PROBLEM DESCRIPTION

The addressed problem is an autonomic conversion of user-driven application requirements to provider-driven network requirements. This conversion is done by first defining a simple graph that represents the application end-points and abstract topological, performance, security, and reliability demands. The graph is then stepwise converted to a VN that represents the physical devices and application virtual nodes with their connectivity, types, locations, and security domains. The connectivity might be simple links or composite links realized by adding chains of network functions that satisfy both application requirements and provider policies, and according to the topologies of the application and physical network. This virtual network is then mapped onto the physical network by first mapping its nodes according to their properties and resource require-

ments. Subsequently, the simple and composite links are mapped using the branching algorithms.

The composite links with SFCs are mapped using a greedy method that checks the remaining length and next available resources of the target path to make placement decisions in advance. This adaptive method spreads or consolidates the VNFs over the path according to its length. Consolidating VNFs on servers saves network bandwidth and reduces the delay, even if the traffic will traverse a long remaining network path after traversing the chain. However, functions that should be placed in different locations/domains cannot be consolidated in one location/domain.

The target use case is remote asset management in industrial environments with three ICT levels: IIoT/factory hall level, edge computing level, and cloud computing level. The used chain mapping method considers the limited resources and dense traffic at the edge computing level. The random topology used for evaluating the chain embedding heuristic reflects such a topology.

3.2.1 *Use Case*

Asset management is the process of tracking the physical assets and making smart decisions based on the gathered data from the assets and environment. Asset management has been listed as one of the five top industrial IIoT use cases by IBM [143]. In asset management, information about assets is actively tracked without any human involvement. Several types of sensors, such as temperature, humidity, pressure, and proximity sensors, are used to gather data that is transferred to edge/cloud computing for making smart decisions.

Three levels of data processing are represented in this use case: factory hall, edge computing, and cloud computing. Furthermore, the factory hall includes an IIoT gateway and several types of devices (such as sensors and cameras) that produce or consume certain rates of data. The use case also represents multiple locations, and each location might include either multiple factory halls and one edge computing center, or a cloud computing center. The edge and cloud computing levels are assumed to host data analysis and decision making functions. Our use case includes three locations with multiple domains that belong to the three mentioned levels, as depicted in Figure 3.1.

The cloud computing level located in a separate location (location 1 here) performs detailed analysis on the gathered data from the manufacturing locations (2 and 3 here). The analysis results are used for making strategical enterprise decisions, for example, related to customized production.

The edge computing level located in each manufacturing location (2 and 3) performs analysis on the gathered data from the factory halls of this location. The analysis results are used to track the locations of assets and employees and environmental data to predict/detect anomalies in real-time, which are mainly related to safety and security. Edge computing can provide local data analysis with accepted latency in order to activate certain actions in real-time using the actuators in the factory hall. These actions are required for cases that cannot be detected by the devices in the factory hall and require data gathering and analysis. For example, detecting physical security breaches by tracking the placement of certain assets. Additionally, because industrial big data are often unstructured, they can be compressed and filtered by edge computing before sending it to the cloud [144].

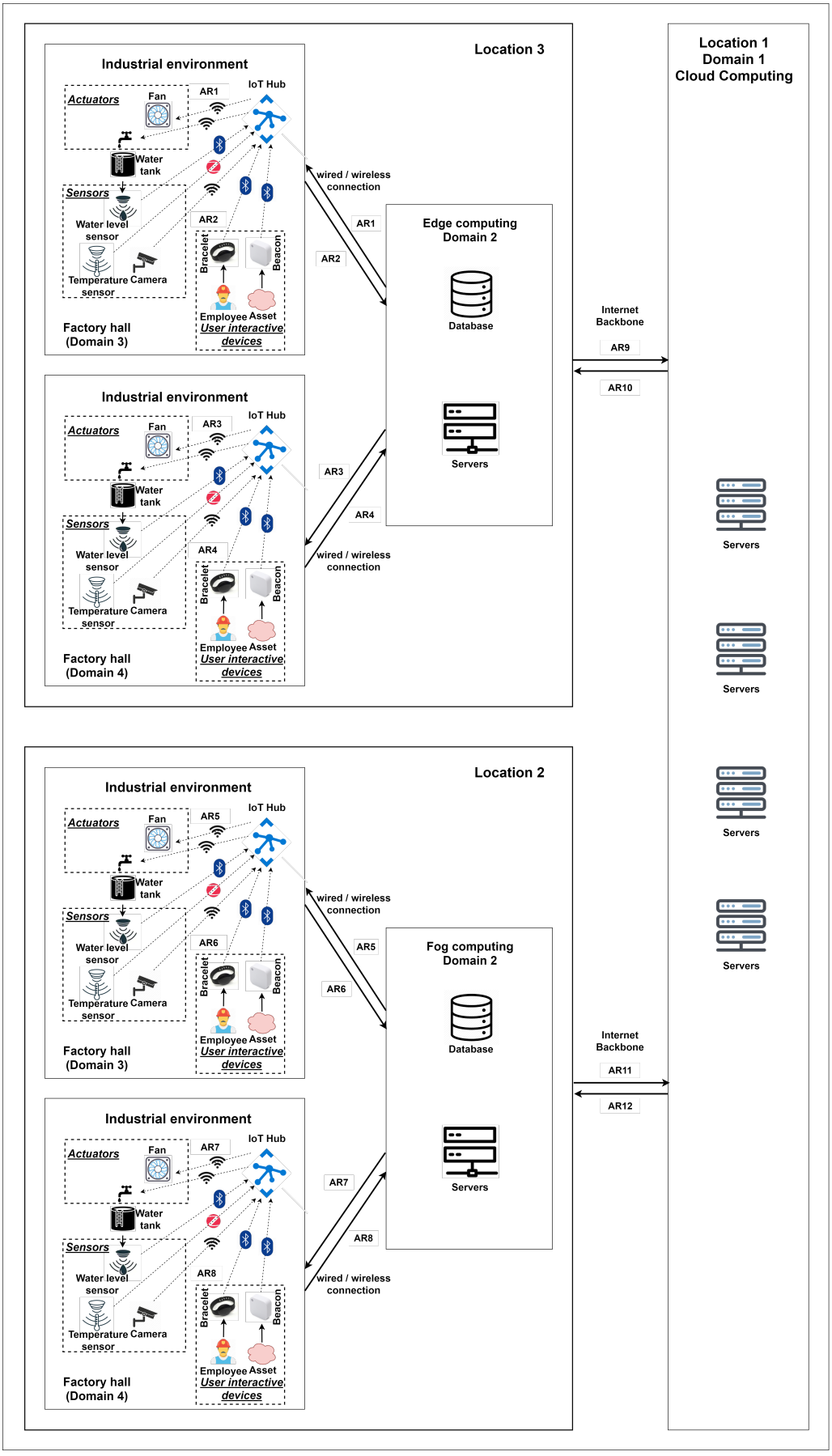

Figure 3.1: Use case [3]

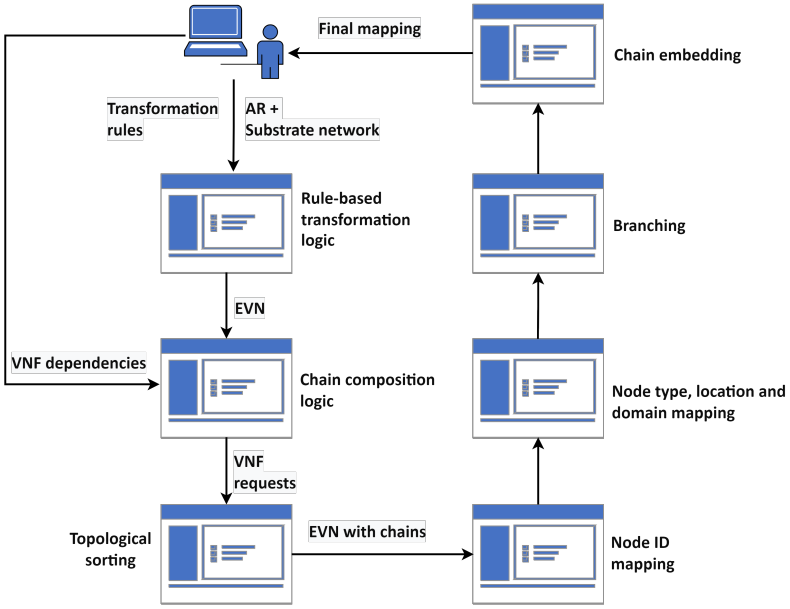

Figure 3.2: System architecture [3]♮

In the *factory hall level*, the IIoT platform allows for easy control and management. The factory halls in locations 2 and 3 might include IIoT hubs, sensors, actuators, cameras, assets with beacons, and employees with Bluetooth low energy bracelets, as shown in Figure 3.1. The environmental information, such as temperature and pressure, are captured by sensors. There might also be cameras in the factory halls to capture images to make decisions through image processing in the edge. For example, facial recognition of employees as an additional security measure and detecting manufacturing problems that cannot be directly detected by the factory sensors or devices. The locations of assets and employees are assumed to be broadcasted by beacons and bracelets. IIoT hubs receive this information and send it to the edge computing in the same location to be analyzed. The actuators might activate/deactivate certain devices (such as fans) either based on local decisions or edge or cloud decisions.

3.2.2 *System Model*

Our main system model (Figure 3.2) includes a system user who provides the AR and SN definition, the transformation patterns and rules, and VNFs' dependencies. The AR is an abstraction level that only defines the application entities and its high-level requirements. It is used to avoid involving the application user in the definition of all required details to realize the application from the networking perspective. The AR can be defined as a directed acyclic graph with VNos and VLis $AR(N_{AR}, E_{AR})$, where N_{AR} is the set of nodes and E_{AR} is the set of edges. The demands are represented as $n^i_{AR}.demandName$ and $e^{ij}_{AR}.demandName$, where $n^i_{AR} \in N_{AR}$ and $e^{ij}_{AR} \in E_{AR}$. The properties are also represented as $n^i_{AR}.propertyName$ and $e^{ij}_{AR}.propertyName$.

We use the same notation for the EVN that results from extending the AR. The SN can be defined as a directed acyclic graph with SNos and SLis $SN(N_{SN}, E_{SN})$, where

N_{SN} is the set of nodes and E_{SN} is the set of edges. The resources are represented as $n^i_{SN}.resourceName$ and $e^{ij}_{SN}.resourceName$, where $n^i_{SN} \in N_{SN}$ and $e^{ij}_{SN} \in E_{SN}$. The properties are also represented as $n^i_{SN}.propertyName$ and $e^{ij}_{SN}.propertyName$.

The AR is transformed into an EVN that might include additional application VNos, VLis, and a VNF request per VLi, where needed. The VNF request defines the required VNFs and their dependencies that represent a mandatory order of VNFs, such as encryption before decryption. The dependencies are predefined in a general VNF graph that defines all available VNFs and their dependencies.

The proposed approach is based on graph transformation [145] in the network virtualization context. The transformation logic applies certain operations (rules) to generate the EVN from a given AR. The transformation is a formalization of a certain policy that is defined by the service provider. It is composed of a pattern and a set of rules. The pattern is a set of conditions to be checked on the input networks, and the rules are the operations to be applied to network entities that match the pattern. These operations are adding, deleting, and updating specific VNos, VLis, VNFs, and demands in the AR and intermediate EVNs. We apply one transformation function per pattern in a predefined order of patterns. A transformation function can take the SN, a pattern P, AR, and intermediate EVN as inputs and return the intermediate/final EVN as output.

After the transformation, we use topological sorting to create all possible FGs for each VNF request based on the VNF dependencies. The FG is also modeled as an acyclic directed graph $FG = (N_{FG}, E_{FG})$ with VNFs n^i_{FG}, and edges that represent data flow e^{ij}_{FG}. Usually, multiple FGs are possible, but one valid FG is selected for each VLi based on location/domain constraints. The embedding logic first maps the application VNos on selected SNos. Subsequently, it finds the mapping paths for each VLi using the branching algorithms. Then it maps each VNF request holding VLi using the chain embedding heuristic, and each simple VLi by verifying its demands.

The problem definition is summarized in the following list and Algorithm 1. The transformations, EVN embedding, and FG embedding are detailed in Section 3.3:

- A physical topology $SN(N_{SN}, E_{SN})$ is defined by the service provider.
- The application requirements $AR(N_{AR}, E_{AR})$ are defined by the user.
- A set of policies (defined by the service provider), where each policy is composed of:
  - A pattern P: conditions on the SN and EVN for checking and comparison of properties, demands, and resources.
  - Transformation T: if P match is found, then perform $T(EVN_{i-1}, SN) = EVN_i$, where $EVN_0 = AR$, and T is a set of rules.
  - A rule R adds or copies VLis or VNos, adds or changes the properties or demands, or adds VNFs to the VNF demand of a VLi.
- The policies are applied with a predefined order on the input AR and SN.
- EVN mapping:
  - For each VLi in the final EVN, if the VLi has a VNF demand, perform topological sorting on the VNF demand:

$$TS(vnfDemand, vnfDependencies) = List\ of\ FGs$$

- For each VLi in the final EVN, if the VLi has a VNF demand, verify the created FGs for location and domain constraints and select a valid FG.
- Map EVN nodes on the SN based on their properties and demands.
- Map EVN simple VLis based on their properties and demands.
- For each VLi with a VNF demand, map a selected FG using the chain embedding greedy heuristic.

Algorithm 1: Main EVN algorithm

Input: $SN(N_{SN}, E_{SN})$, $AR(N_{AR}, E_{AR})$
$EVN = AR$
for $policy^m \in policies$ **do**
  if $policy^m.pattern = True$ **then**
    for $R \in policy^m.transformation.rules$ **do**
      $R(EVN, SN) = EVN$
    end
  end
end
for $n^i_{AR} \in EVN.nodes$ **do**
  $Map(n^i_{AR}) = n^k_{SN}$
end
for $e^{ij}_{AR} \in EVN.links$ **do**
  if $e^{ij}_{AR}.vnfDamand = NULL$ **then**
    $Map(e^{ij}_{AR}) = Branching(e^{ij}_{AR}.source, e^{ij}_{AR}.destination)$
  end
  else
    $FG = AllTopologicalSort(e^{ij}_{AR}.vnfDamand)[0]$
    $paths = Branching(e^{ij}_{AR}.source, e^{ij}_{AR}.destination)$
    $ChainEmbedding(paths, FG)$
  end
end

Figure 3.3 shows an exemplary scenario of the main EVN algorithm. The SN includes two locations with compute instances (servers), where location 1 includes sensors with specific types and data loads. The AR requests connecting all sensors from location 1 to a compute instance from location 2. The node type transformer adds VNos and VLis that represent all matching sensors and adds an application gateway that connects them to the compute instance in location 2. The application gateway is allowed to be mapped on an IoT Hub. The security transformer adds the required security VNFs to the VLi between the locations. Then, the bandwidth and CPU transformers adapt the demands of all VLis and VNos (including those inside the security SFC here) accordingly. Finally, the EVN embedding algorithm maps the VNos and VNFs according to locations and maps the VLis accordingly.

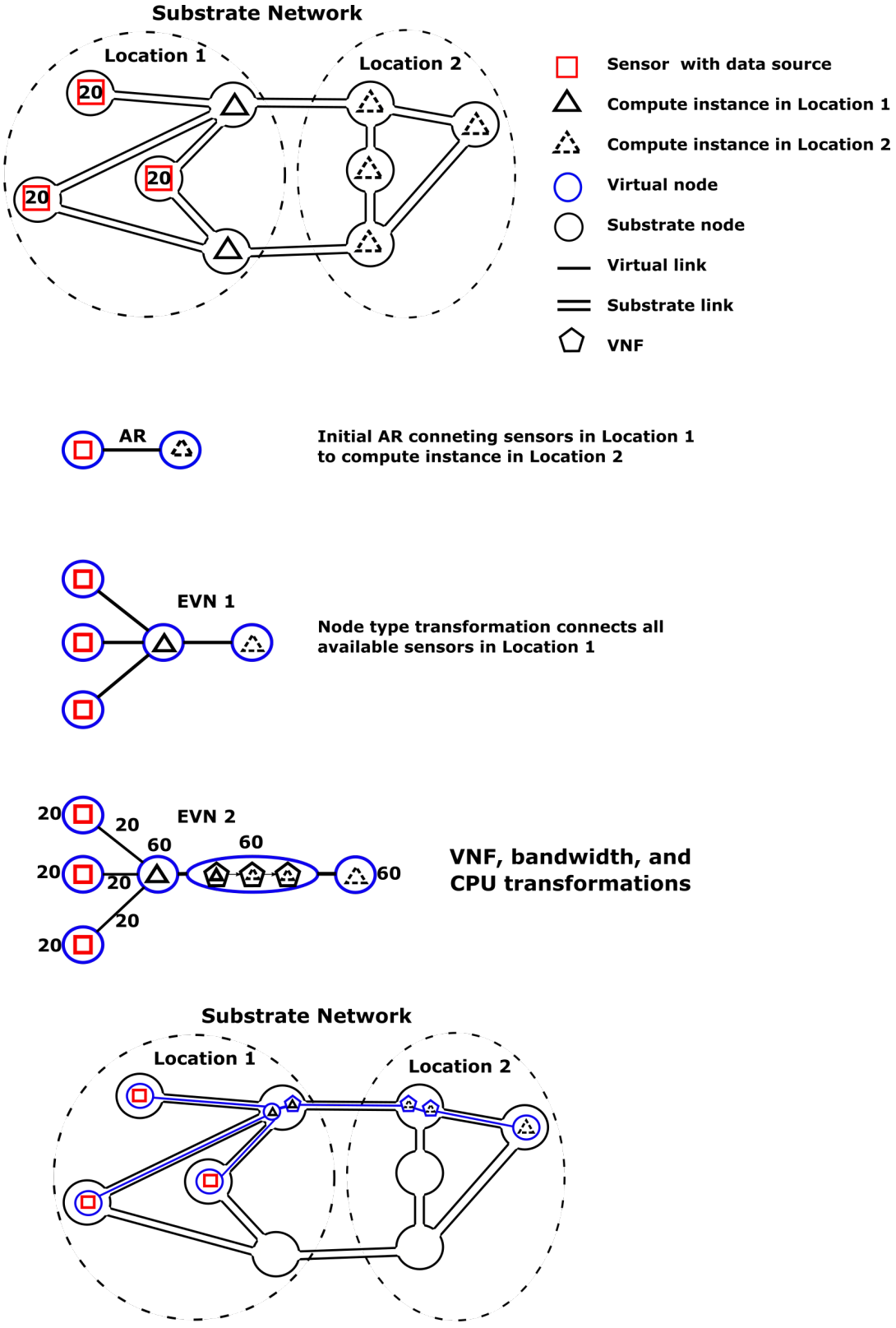

Figure 3.3: An exemplary scenario of the main EVN algorithm

3.2.3 *General Dependency Graph*

The generic dependency graph includes all possible VNFs, as depicted in Figure 3.4, where NM = network monitoring, ENC = encryption, DEC = decryption, FW = firewall, and DPI = deep packet inspection. The similar VNFs distinguished by "source" and "destination" represent the placement of the same functions in different domains/locations according to the traffic direction. The transformations add these security functions only for cross-domain SFCs [5]‡. The arrows show the dependencies between VNFs, which is a transitive relation. For instance, the arrow from the source firewall to the encryption function means that the firewall VNF depends on the encryption VNF and must, therefore, be executed after it. Note here that the dependencies are in the reversed direction from the assumed traffic direction.

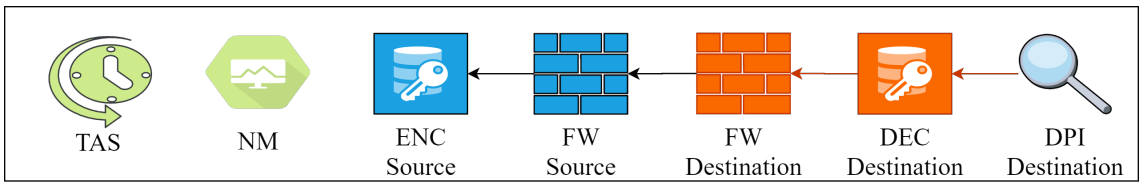

Figure 3.4: General dependency graph [3]‡

Furthermore, the network monitoring VNF is assumed to have no dependencies to create a scenario in which several FGs are possible. The TAS VNF is a latency reduction function (see Section 3.3.1.4) without dependency and it shall be allocated on each server hosting the FG. Therefore, the FG composition method only adds it to the beginning of the FG. We assume that deploying a virtual TAS on a host means adding its functionality to all other VNFs that should be customizable to perform multiple tasks. The multi-objective goal here means that we can perform security and scheduling functions by one VNF. We note here that using virtual TAS in real environments requires either using only standard servers as network hardware or TAS-capable network hardware with the deployment of a schedule synchronized with the TAS SFC schedule. Figure 3.5 presents the FGs that are possible from the general dependency graph.

3.3 METHODOLOGY

In this section, we detail the graph transformation methods used to compose the EVN, the FG composition, EVN embedding, and FG embedding algorithms. Furthermore, we present a detailed calculation of the complexity of the whole approach in addition to the branching algorithms from Chapter 5.

3.3.1 *Transformations*

The following transformations are applied with the same order on the AR and SN. This order reflects a service provider general policy. First, the redundancy policy is only applied to the main VLis in the AR. Second, the target devices are added based on type, location, and domain. Third, the security VNFs are added to cross-location and cross-domain VLis. Fourth, a low latency specific VNF is added to the chain if needed. Fifth, the bandwidth is updated for all VLis based on data sources and bandwidth

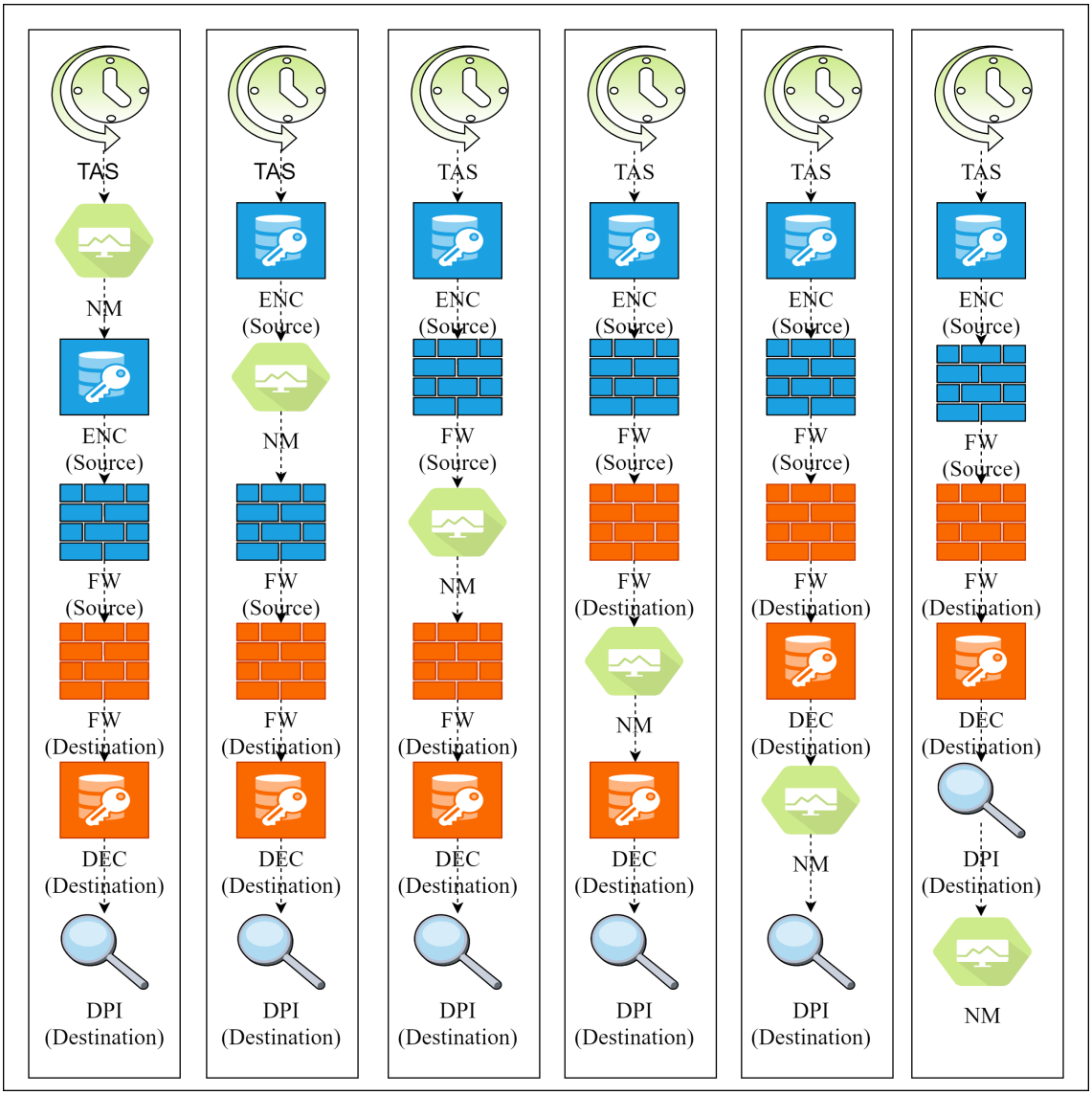

Figure 3.5: Possible FGs for the general dependency graph [3][‡]

multipliers, then the CPU demands based on bandwidth demands. Finally, the required load balancing at a destination application VNo is added. For each transformation, the whole SN is checked for the pattern match, and a new EVN is created and given to the next transformation.

3.3.1.1 *Redundancy*

This transformation copies the VLi with redundancy demand. Original and new VLis are both provided with a property "redundant link" that holds an identifier of the backup link. This is used, for example, by the mapping algorithm to find the most disjoint path (maximal branching) for the second link from the k shortest paths between the source and destination, found by Dijkstra algorithm. The chain embedding forces the most disjoint paths, even if the VLi holds a VNF demand. However, the details of the reliability-aware

minimal and maximal branching including the reliability values are left to Chapter 5 to reduce the complexity in this chapter.

$$P : \{e^{ij}_{AR} | e^{ij}_{AR}.redundancyDemand = True\}$$
$$T(SN, AR) : \{AR \cup copy(e^{ij}_{AR})\}$$

3.3.1.2 *Nodes Types*

A single VNo in the AR might represent multiple required devices in the SN. For example, the administrator might need to include all of the sensors from a certain type in a factory hall in an AR, but this is very complex. This transformation adds to the EVN the VNos that represent certain physical devices using the ID demand. The ID demand in VNE and ALEVIN forces a certain VNo to be mapped on a certain SNo. This mapping is required to create the respective VLi and map it. This transformation is applied on each VNo and the pattern match is checked for each SNo. A new VNo is added to the AR with an ID demand that matches the ID of the compared SNo if the types of both are "device", and the subtypes, locations, and domains are equal. Subsequently, a VLi between the new VNo and the original VNo is added.

The new VNo and VLi are copies of the original to hold the same requirements. However, in our SN definition format, we attach a data resource to all devices with the parameters of cycle time and frame size. These parameters are copied to the matching new VNo to calculate its bandwidth demand by the bandwidth transformation. Here, we assume that each VNo defined in the original AR to represent a physical device has one outgoing edge. When the pattern is checked for all SNos, the original VNo is converted to type "application" and subtype "gateway". This method will force mapping the original VNo that represents a set of devices to a gateway.

$$P : \{n^k_{SN}.type = "device" \wedge n^j_{AR}.type = "device" \wedge n^k_{SN}.subtype = n^j_{AR}.subtype$$
$$\wedge n^k_{SN}.location = n^j_{AR}.location \wedge n^k_{SN}.domain = n^j_{AR}.domain\}$$
$$T(SN, AR) : \{AR \cup n^i_{AR} | n^i_{AR} = copy(n^j_{AR}), n^i_{AR}.ID\_Demand = n^k_{SN}.ID,$$
$$n^i_{AR}.dateDamand = n^k_{SN}.dateResource\}\{AR \cup e^{ij} | e^{ij} = copy(e^{jl})\}$$

After T is applied to all SNos:

$$\{n^j_{AR}.type = "application", n^j_{AR}.subtype = "gateway"\}$$

3.3.1.3 *Security VNFs*

In this transformation, we assume that the service provider defines certain security policies. For example, for a cross-location or cross-domain VLi, an SFC with certain security VNFs must be attached to this VLi. Our transformation policies presented here consider both cases. For cross-location VLis, encryption, firewall, deep packet inspection, and monitoring VNFs are added to the VNF demand of the VLi. We assume that encryption is not needed for cross-domain VLis in the same location. Except for the network monitoring function, each type of VNF is attached to the source or destination location/domain. Specific location/domain demands are attached to the added VNFs

based on the location/domain demands of the AR. However, the general dependency graph forces the order of these VNFs in the FG. The VNFs in the link FG are assigned the node type "VNF".

$$P : \{e^{ij}_{AR} | n^{i}_{AR}.location \neq n^{j}_{AR}.location\}$$

$$T(SN, AR) : \{e^{ij}_{AR}.vfnDemand.add(ENC, FWS, FWD, NM, DEC, DPI)\}$$

$$P : \{e^{ij}_{AR} | n^{i}_{AR}.location = n^{j}_{AR}.location \wedge n^{i}_{AR}.domain \neq n^{j}_{AR}.domain\}$$
$$T(SN, AR) : \{e^{ij}_{AR}.vnfDemand.add(FWS, FWD, NM, DPI)\}$$

3.3.1.4 *Low Latency VNF*

When a VLi in the AR has a low latency demand, a specific type of VNF is added to its VNF demand, virtual TAS [2]‡, which can reduce the latency of traffic processed by servers by scheduling it based on its cycle and load. However, the order of this VNF in the chain is based on the mapping results. This means that it shall be instantiated at each server that hosts the VNF demand. For this reason, the chain composition algorithm executed after the transformations adds this VNF to the beginning of the FG. Subsequently, the EVN embedding algorithm maps this VNF on each server as the last VNF from the FG.

$$P : \{e^{ij}_{AR} | e^{ij}_{AR}.latencyDemand = True\}$$
$$T(SN, AR) : \{e^{ij}_{AR}.vnfDemand.add(TAS)\}$$

3.3.1.5 *Bandwidth*

Adding VNos, VLis, and VNFs to the EVN requires the adaptation of the resource requirements. For example, adding a VNo that represents a certain device requires adapting the bandwidth demand of the new and original VLis based on the parameters of the data resource of the device. Subsequently, all of the next VLis shall be updated, including these inside the FG. This approach can be applied to feed-forward VNs, which we assume in our use case and use in defining our exemplary ARs.

We consider two types of VNos, VNos that have no incoming edges and intermediate VNos that have incoming edges. For each VNo that has no incoming edges, its produced load per interval, if exists, is used to calculate the required bandwidth that is assigned to all of its outgoing edges.

$$P : \{e^{ij}_{AR} | e^{ij}_{AR}.bandwidthDemand = 0 \wedge n^{i}_{AR}.inEdges = 0\}$$
$$T(SN, AR) : \{e^{ij}_{AR}.bandwidthDemand =$$
$$n^{i}_{AR}.dataDemand.size / n^{i}_{AR}.dataDemand.cycle\}$$

For intermediate VNos that have incoming edges, we sum the required bandwidth of all incoming edges and set it as demanded bandwidth for all outgoing edges. If the VNo has the property of multiplier, then it is considered during the calculation for the

outgoing bandwidth demand. In our work, we define a specific bandwidth multiplier for each specific VNF type as a property. The default value is 1.

$$P : \{e^{ij}_{AR} | e^{ij}_{AR}.bandwidthDemand = 0 \wedge \exists e \in n^i_{AR}.inEdges : e.bandwidthDemand \neq 0\}$$

$$T(SN, AR) : \{e^{ij}_{AR}.bandwidthDemand =$$

$$\sum_{k=0}^{n^i_{AR}.inEdges} (e^{ki}_{AR}.bandwidthDemand \times n^i_{AR}.bandwidthMultiplier)\}$$

3.3.1.6 *CPU and Load Balancing*

After adapting the bandwidth demands, we adapt the CPU demands of the VNos and VNFs using the CPU factor property, which is used to determine the CPU demand from the incoming bandwidth.

$$\{T(SN, AR) : n^i_{AR}.CPUDemand = n^i_{AR}.incomingBandwidth \times n^i_{AR}.CPUFactor\}$$

When the resulting CPU demand of a VNo is larger than the minimum CPU capacity of all application SNos (servers) in the same domain, this VNo is copied k times, where k is the ratio of the CPU demand and minimum CPU capacity. The original VNo is converted to type *"loadBalancer"* and a link between it and each new VNo is created. This is a form of coordination between the EVN composition and embedding stages to avoid the re-transformation of the AR to an EVN as a result of a failed mapping in the embedding stage due to a lack of resources. However, and for simplicity, we currently only apply this to application VNos with incoming edges and no outgoing edges. This exemplary scenario represents a data analyzer that is an application end-node in the AR.

$$P : \{n^i_{AR}.type = "application" \wedge n^i_{AR}.inEdges > 0 \wedge n^i_{AR}.outEdges = 0$$

$$\wedge n^i_{AR}.CPUDemand > (C = min(n^j_{SN}.CPUResouce | n^j_{SN}.domain = n^i_{AR}.domain))\}$$

$$T(SN, AR) : \{AR \cup \{n^1_{AR}, ...n^l_{AR}, ...n^k_{AR} | n^l_{AR} = copy(n^i_{AR}),$$

$$n^l_{AR}.CPUDemand = \frac{n^i_{AR}.CPUDemand}{k}, k = \frac{n^i_{AR}.CPUDemand}{C}\}\}$$

$$\{AR \cup \{e^{i1}_{AR}, ...e^{il}_{AR}, ..., e^{ik}_{AR} | e^{il}_{AR}.bandwidthDemand = \frac{n^i_{AR}.incomingBandwidth}{k}$$

$$n^i_{AR}.subtype = loadBalancer\}\}$$

3.3.2 *Creating FGs Using Topological Sorting*

In the chain composition stage, the possible FGs are calculated. The authors in [41] introduce a context-free grammar to formalize the request. Topological sorting is a known graph method to sort a directed acyclic graph. This method is used in several works on SFC composition, such as [47] and [146], in order to generate the FG. Our proposed approach is also based on topological sorting, but we calculate all possible FGs. Based on [147], the topological ordering of a graph $G = (V, E)$: $ord : V \rightarrow 1...n$ for $n = |V|$ exists, if $\forall (v, w) \in E$: $ord(v) < ord(w)$. For each acyclic graph, a topological ordering exists, and each graph that has a topological ordering is acyclic.

Using topological sorting for the chain composition enables us to prove whether the VNF request is cycle free and if a possible FG can be calculated. It might happen that the dependencies among VNFs are not defined correctly by the service provider leading to cyclic VNF requests. For example, for a VNF request including encryption and decryption, where both functions depend on each other, no valid chaining is applicable due to the dependency loop. A simplified approach for calculating the topological ordering is to iterate over all VNos and taking a VNo v in each iteration, where $inDegree(v) = 0$, and adding it to an array. Afterward, v and its outgoing edges are removed from the graph. This operation is repeated until there is no VNo left with $inDegree = 0$. If the graph is empty, then a topological ordering exists, otherwise it is not cycle free. The associated pseudo-code, which is based on [147], is presented in Algorithm 2.

Algorithm 2: Topological sorting

```
1 TopologicalSort (G)
2     index = 0
3     ordering[]
4     while G has at least one node n where inDegree(n) = o do
5         ordering[index] := n
6         index++
7         G := G − {n} ;                    // Remove node and its outgoing edges from G
8     end
9 end
```

In the context of chain composition under the location and domain constraints, it is essential to calculate all possible sortings for a given graph G. Then, the resulting FGs are verified against the domain and location constraints. The verification checks the consistent order of domains and locations in the FG. This means that the first continuous part of the FG shall belong to the source domain/location, and a second continuous part of the FG shall belong to the destination domain/location. Then, a valid FG is selected for mapping the SFC. To the best of our knowledge, this method of finding all sortings and verifying FGs based on location and domain constraints is not addressed by the NFV community.

For this purpose, the above-mentioned pseudo-code has to be adapted to calculate another ordering when the graph has multiple VNos where the $inDegree = 0$. Algorithm 3 represents a pseudo-code for calculating all of the possible sortings for a given graph. This Algorithm recursively calculates all possible orderings for a given graph G. The algorithm iterates over all VNos in the graph. For each VNo, if $indegree = 0$, it is added to the tentative result and removed with all outgoing edges from the graph. If the graph is not empty, the function is recursively called until all VNos are visited or the graph is empty, then the tentative result is assumed to be a valid ordering and is added to the result.

3.3.3 *EVN Embedding*

EVN embedding is the combination of VNE and NFV resource allocation, and it can be separated into two stages. The first stage is to embed all of the nodes and simple VLis, and the second stage is to embed the VLis with a VNF request. The node mapping first considers the ID demand to map devices (specific VNo on a specific SNo). Subsequently, the type, subtype, location, and domain are considered for mapping the

Algorithm 3: Finding all topological sortings

```
1  ordering[]
2  AllTopologicalSort (G, tentativeResult)
3  |  G' = copy(G)
4  |  while G' has at least one node n where inDegree(n) = 0 do
5  |  |  tentativeResult' = tentativeResult
6  |  |  tentativeResult'.add(n)
7  |  |  G' := G' − {n}
8  |  |  if G' = ∅ then
9  |  |  |  ordering ∪ tentativeResult'
10 |  |  else
11 |  |  |  AllTopologicalSort (G', tentativeResult)
12 |  |  end
13 |  end
14 end
```

other nodes. For the nodes of type *"application"* and subtype *"computeInstance"*, the CPU demand/resource is considered.

The link mapping uses the branching algorithms that use Eppstein's algorithm [148] for finding k shortest paths between the source and destination VNos of the VLi. For the simple link mapping (without VNF demand), the candidate paths are additionally verified for the bandwidth demand. For mapping the VLis with VNF demand (VNF request), the candidate paths are given to the chain embedding heuristic. In both cases, the redundant VLi is mapped on another candidate path from the k shortest paths. This path is chosen based on the branching policy used, for example, as the path with the least number of common SNos with the path used for the original VLi.

3.3.4 *Chain Embedding Heuristic*

In [47], the authors assume that each SNo on the path can host one VNF. The algorithm presented in [46] is based on backtracking and, in the case of rejection, tries a different path starting from the last successfully embedded VNo. Finding a path where the number of hops exactly matches or is greater than the number of VNFs might be impossible or inefficient, in particular with our use case. Our proposed embedding algorithm tries to find a solution based on the FG or path length. It is assumed that a SNo can host multiple VNFs when the resource capacity allows. Based on this assumption, a path with fewer hops than the FG can be utilized. On the other hand, for a path longer than the FG, the algorithm tries the next hop on the path in case of lack of resources. Algorithm 4 represents the pseudo-code for our proposed algorithm.

As input, a path connecting two VNos embedded onto different SNos and the FG are given. Initially, the algorithm takes the first nodes in the FG and the path. If the target VNF can be allocated on the current SNo, then the algorithm verifies if the number of residual VNFs is smaller or equal to the residual hops on the path. If possible, the algorithm tries to embed multiple VNFs on a target SNo when the residual path length is less than the residual FG length. This is intended to find a valid embedding to avoid backtracking. If available, the next FG's node and the next hop on the path are verified.

Algorithm 4: Greedy chain embedding algorithm

```
1  ChainEmbedding (path, FG)
2      path ;  // Including all nodes (source, destination, and intermediate nodes)
        and links on the path
3      T_SN = path.getFirst()
4      for n^i_FG ∈ FG do
5          if verify(n^i_FG, T_SN) then
6              mapping(n^i_FG, T_SN))
7              if residualChainLength <= residualPathLength AND verify(n^{i+1}_FG, path.next()) then
8                  T_SN = path.next()
9              end
10         else
11             while (fulfilled = false) OR (path.next() ∈ SN) do
12                 fulfilled = verify(n^i_FG, T_SN)
13                 if fulfilled then
14                     mapping(n^i_FG, T_SN))
15                     if residualChainLength <= residualPathLength AND verify(n^{i+1}_FG,
                        path.next()) then
16                         T_SN = path.next()
17                     end
18                     break;
19                 end
20                 T_SN = path.next()
21             end
22             if fulfilled = false then
23                 Reject request;
24             end
25         end
26         verifyResidualPath(dr_out, residualPath) ;     // If available, verify residual
            links on the path with respect to the bandwidth output of the last
            node in the FG
27     end
28 end
```

If the verification is successful, then the target SNo is set to the successor of the current one. This verification is intended in this context to prevent setting the target SNo to the next SNo if it might not be able to fulfill the requirements. If the residual FG is shorter than the residual target substrate path, we try a balanced embedding over all SNos on the path to avoid consolidation. However, because the path might include SNos from different domains/locations or can not host the next VNF, it is necessary to verify whether the next SNo is able to provide enough resources and fulfills the special requirements of the next FG element.

In the next step, the algorithm takes the next VNo in the FG and verifies whether a successful allocation is possible. In case of no possible allocation, it tries to verify whether the next SNo on the path can fulfill the requirements and, otherwise, it tries the next SNo until the last SNo is reached. In addition to VNo verification, the links inside FGs are verified against the SLis. When the last VNF is not mapped to the destination SNo on the path, then the outgoing data rate dr_{out} of the last VNF has to be verified over the residual path.

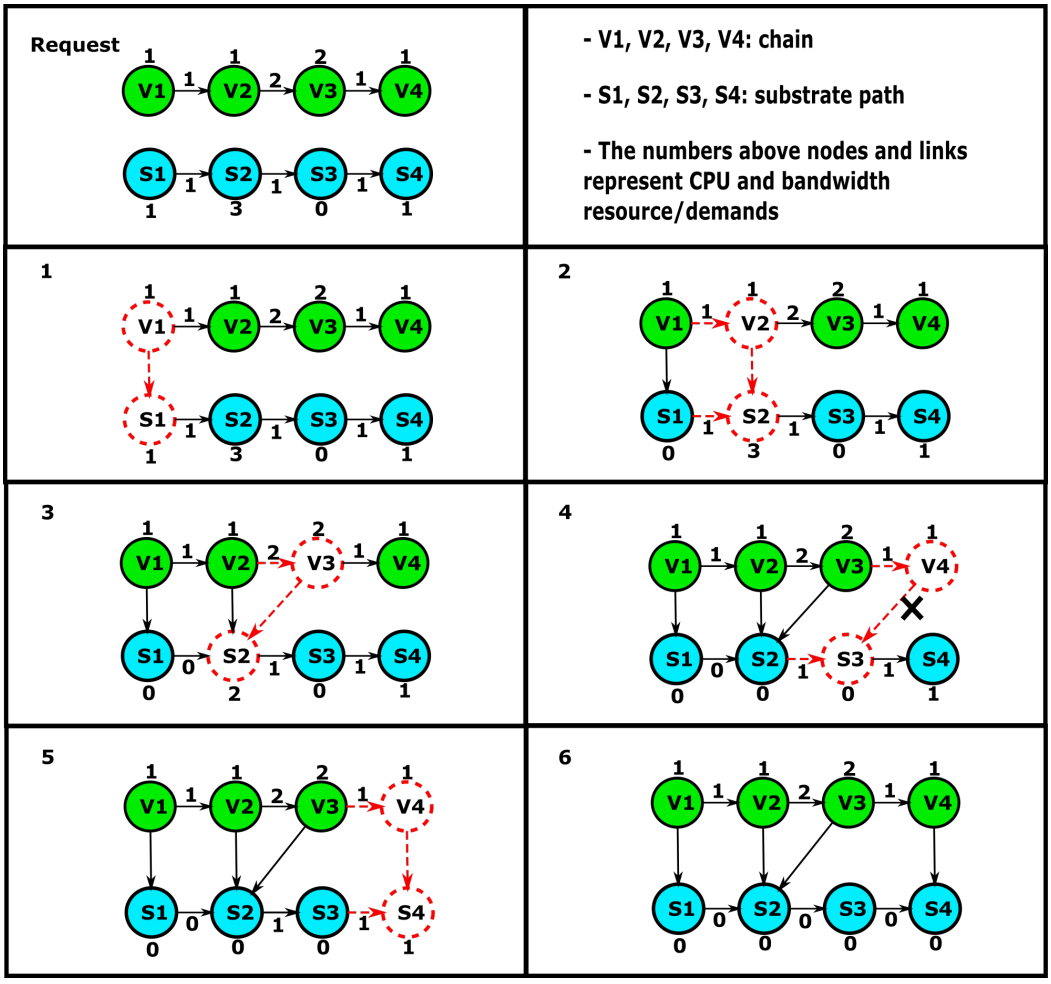

Figure 3.6: FG embedding example [3][♮]

Figure 3.6 presents an example of the FG embedding. The initial FG contains four VNos *(V1, V2, V3, V4)* and the target substrate path has four SNos *(S1, S2, S3, S4)*. Resources and demands are assigned to nodes and links in both the FG and substrate path. Nodes and links currently under investigation are dotted. The embedding starts by checking whether *V1* can be hosted on *S1*. Because enough resources are available, the assignment is made here and the new target SNo is set to *S2*. Again, the algorithm verifies whether *V2* can be embedded on *S2* and performs the assignment. The target SNo remains *S2* since the link bandwidth demand between *V2* and *V3* cannot be fulfilled on the link between *S2* and *S3*.

In the next step, VNo *V3* is assigned to *S2* and the target SNo is set to *S3*. Next, the algorithm verifies whether *V4* can be hosted on the target SNo *S3*. In this case, it is not possible since *S3* has no enough CPU resources. Therefore, the algorithm keeps *V4* as the target FG VNo and verifies whether the next SNo on the path *S4* provides sufficient resources. *S4* provides enough resources and the assignment is performed. Because all FG VNos are embedded, the algorithm returns true and terminates. For simplicity, this example considers resource capacities and not domains and locations.

The presented example shows that assuming the same length of the FG and substrate path is not reasonable. It might be required in certain scenarios to embed multiple VNFs

on a single SNo in order to find a valid solution. The proposed algorithm performs an embedding while taking link and node constraints into account through forward verification.

3.3.5 *Complexity of the Proposed Methods*

We first make the following assumptions to estimate the complexity of the proposed algorithms:

- Neglecting the operations applied by the rule on only a small subset of the EVN entities.
- Considering that the maximum size of the final EVN (nodes M, links L) is known and used to estimate the complexity even for intermediate EVNs.
- (D, D) is the size of the dependency graph. We consider that, in NFV, a function typically has one dependency, so the number of edges in the dependency graph can be approximated to the number of VNFs. The size of the dependency graph is considered as the size of each VNF demand.
- P is the diameter of the SN graph (N,E), which is the number of edges in the shortest path between the most distant vertices. We use this worst-case value as the length of the shortest path on which the chain will be embedded.
- We are calculating the complexities in the worst cases without considering locations and domains that might reduce the search space for node and link mapping, in particular, in specific topologies, like in our use case.

We estimate first the complexity of each step in our system model:

- Transformations:
  - Redundancy – *O(L)*: checking the redundancy demand of all EVN links and copying them when needed.
  - Node types - *O(N.M)*: comparing certain properties of SNos and VNos and adding the required entities to the EVN with their demands.
  - Security – *O(L)*: comparing the source and destination locations/domains of the EVN links and adding the required VNFs.
  - Low latency – *O(L)*: checking the latency demand of the EVN links and adding the TAS VNF where needed.
  - Bandwidth – *O(L)*: checking the bandwidth demand of each VLi and adjusting it based on the bandwidth demands of the incoming edges to its source VNo.
  - CPU - *O(M)*: adjusting the CPU demand of all EVN nodes based on the incoming bandwidth, cloning certain nodes, and then adding a load balancing function.
  - Total complexity of the transformations: $O(4L+M\,(1+N)) = O(L + M.N)$, when considering that $N \gg 1$.

- Topological sorting and FG verification: the complexity of finding all topological sortings depends in the worst case on the number of permutations in the group of VNF types D (*D!*). In our work, we define 7 VNF types and this value is 5040. The verification of each FG takes D steps. In the worst case, these operations shall be performed for each VLi. The worst-case complexity is *O(L.D.D!)*. However, we simplified this method and reduced the effect of D by integrating the FG verification with finding the topological sortings. In our scenario, we can find a valid FG after few sortings, and we can neglect the effect of D to reduce the real complexity to *O(L)*.

- Node mapping – *O (N.M)*: finding the SNo with the matching ID, properties, or resources of each VNo.

- Chain embedding - *O (2P)=O(P)*: when mapping a VNF is tried on a SNo, the demands are compared to the resources. A VNF can be mapped on the latest SNo used from the path or any other previous node. In the worst case, all path SNos are checked twice for each placed VNF and next VNF that cannot be placed on the same SNo, so a following one is checked. The P value varies with the SN size.

- Link mapping: the complexity of the k-shortest path algorithm is *O(E+N.log N))* [148]. This algorithm is used for mapping all VLis. The branching methods in Chapter 5 process each pair from the k-shortest paths and check the number of common nodes, and check the reliability of the path pair. Finding the pairs is equal to finding combinations (k,2), which has the complexity of $k!/2(k-2)! = k.k-1/2$. Since k is small, this value is small and has no effect on the O notation. For example, if we are selecting 5 paths, this value is 10. For each pair, the nodes of the paths are compared, with the complexity of $P!/2(P-2)! = P.P-1/2 = O(P^2)$. We assume for the worst case that all VLis have VNF demands for which chain embedding will be performed. The total complexity of link mapping is: $O(L.(E+N.logN+P+P^2)) = O(L.(E+N.logN+P^2))$. In tree-like topologies like in our use case, $P \ll E$, and the complexity of the link mapping is *O(L.(E+N.log N))*.

The total complexity of the solution is:

$$O(L+N.M+L+N.M+L.(E+N.logN)) = O(N.M+L.(E+N.logN))$$

3.4 IMPLEMENTATION

ALEVIN [11] is an open-source framework written in JAVA and used to develop, compare, and analyze VNE algorithms. The SN and VN are represented as directed or undirected graphs in which the network entities hold demands and resources that represent either consumable resources or properties, such as ID. In this work, we add type, location, domain, redundancy, latency, and VNF demands and resources. The embedding of each node/link in the VN only succeeds when there are enough resources and matching properties in SN nodes/paths. The Visitor Pattern method is used in ALEVIN to map demands on the respective resources and to occupy (reserve) consumable resources.

An extension with NFV support is available for ALEVIN and developed in [46]. However, we developed our own NFV extension that fits our system model and transformation methods. We added a generic structure to ALEVIN to define transformations that are based on an abstract transformation class and method that takes the VN and SN as input.

ALEVIN implements several node and link mapping algorithms, including coordinated node and link mapping. It also implements several evaluation metrics, such as runtime, admission ratio, and cost. However, ALEVIN is highly flexible for adding new algorithms and metrics. It also provides an evaluation framework, in which different forms of random topologies can be defined with specific parameters. This framework also enables the definition of used mapping algorithms, metrics, and ranges of values for consumable resources and demands. ALEVIN supports multiple data formats, mainly Extensible Markup Language (XML), for representing the SN and VN and mapping the results.

For fixed topologies that represent our use case, we use the easier JavaScript Object Notation (JSON) format to represent the AR, SN, and mapping results. For this purpose, we developed the required parsers that convert the JSON structures to the respective networks, entities, and resources and demands. The AR JSON definition format defines the application end-points and connectors. For each node, an ID, name, type, subtype, location, domain, and, if applicable, CPU demand are defined. For the connector, the source and destination nodes are defined, and whether redundancy and low latency are required. The SN JSON definition format is similar but includes resources instead of demands, data resources with cycle and size, and bandwidth resources for SLis. Listings 3.1 depicts an exemplary definition of an AR from the use case in Figure 3.1. This AR connects the temperature sensors from a factory hall to a compute instance in the edge computing level.

Listing 3.1: JSON format for AR definition

```
{
    "applicationRequest": {
        "applicationEndPoints": [
        {
            "id": "1",
            "name": "src",
            "demands": [{
                "name": "typeDemand",
                "demandedNodeType": "device",
                "demandedNodeSubType": "temperatureSensor",
                "function": "temperatureSensor",
                "location": "Location3",
```

```
                "domain": "Domain3"
            }]
        },
        {
            "id": "2",
            "name": "dst",
            "demands": [{
                "name": "typeDemand",
                "demandedNodeType": "application",
                "demandedNodeSubType": "computeInstance",
                "function": "dataAnalyzer",
                "location": "Location3",
                "domain": "Domain2"
            },
            {
                "name": "cpuDemand",
                "demandedCPU": "20"
            }]
        }],
        "applicationConnectors": [{
            "id": "1",
            "name": "link1",
            "srcNode": "1",
            "dstNode": "2",
            "demands": [
                    {
                        "name": "redundancyDemand",
                        "parameters": []
                    }
                    {
                        "name": "lowLatencyDemand",
                        "parameters": []
                    }
            ]
        }]
    }
}
```

Our SN topology that represents the use case is depicted in Figure 3.7. Listing 3.2 depicts the map of the SN represented by our JSON structure, which is only partially showed for simplicity. The JSON definition format of the mapping results is further detailed in Section 3.5.1.

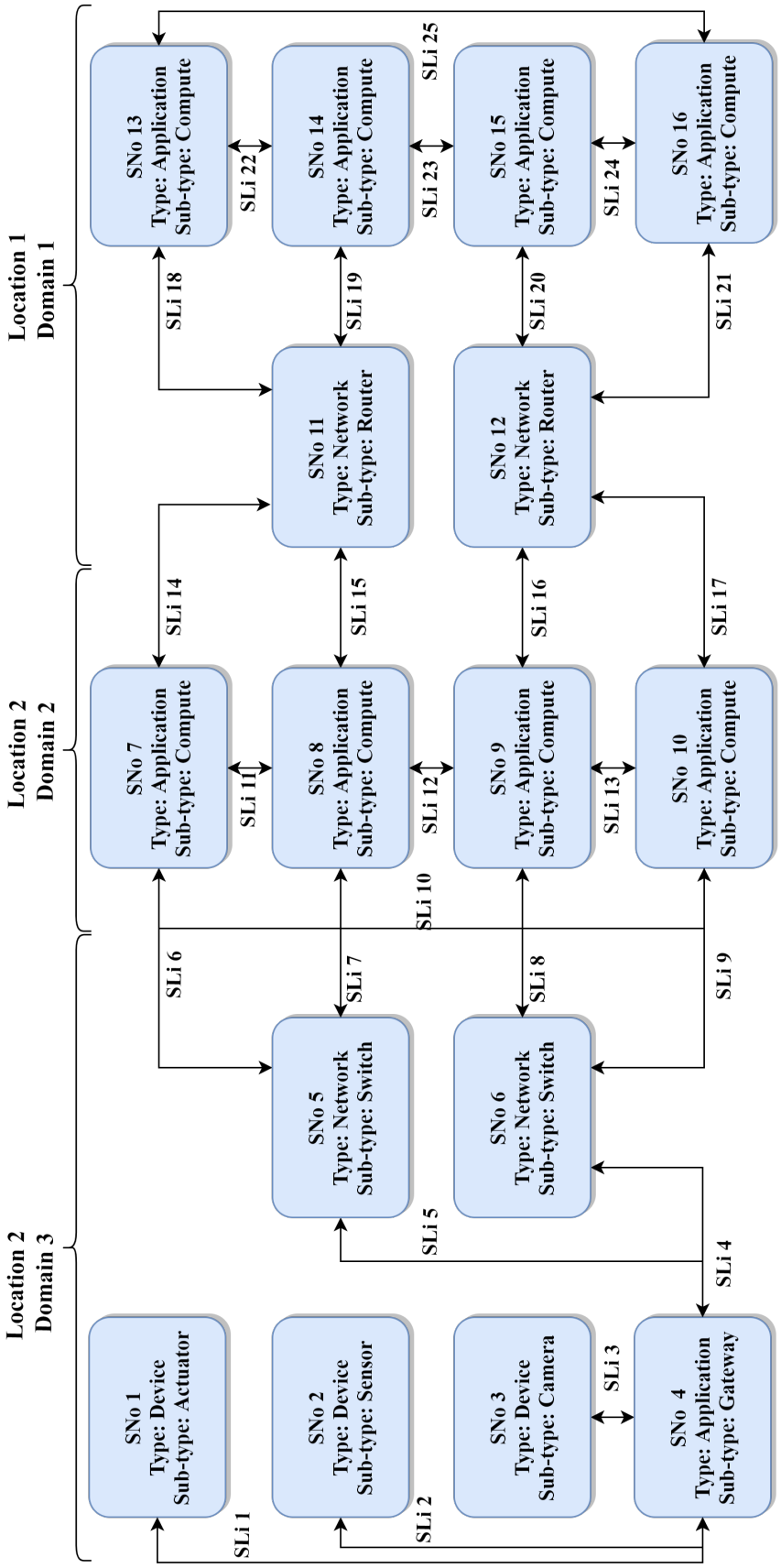

Figure 3.7: Substrate network topology

Listing 3.2: JSON format for SN definition

```
{
    "substrateNetwork": {
        "substrateNodes": [
        {
            "id": "2",
            "name": "SNo 2",
            "resources": [{
                "name": "typeResource",
                "nodeType": "device",
                "nodeSubType": "temperatureSensor",
                "function": "temperatureSensor",
                "location": "Location2",
                "domain": "Domain3"
            },
            {
                "name": "dataResource",
                "cycle": "1",
                "size": "20"
            }]
        },
        {
            "id": "4",
            "name": "SNo 4",
            "resources": [{
                "name": "typeResource",
                "nodeType": "application",
                "nodeSubType": "gateway",
                "function": "gateway",
                "location": "Location2",
                "domain": "Domain3"
            },
            {
                "name": "cpuResource",
                "cycles": "1000"
            }]
        },
        {
            "id": "6",
            "name": "SNo 6",
            "resources": [{
                "name": "typeResource",
                "nodeType": "network",
                "nodeSubType": "switch",
                "function": "switch",
                "location": "Location2",
                "domain": "Domain3"
            }]
        },
        {
            "id": "8",
            "name": "SNo 9",
```

```
            "resources": [{
                "name": "typeResource",
                "nodeType": "application",
                "nodeSubType": "computeInstance",
                "function": "computeInstance",
                "location": "Location2",
                "domain": "Domain2"
            },
            {
                "name": "cpuResource",
                "cycles": "3000"
            }]
        }
        ],
        "substrateLinks": [{
            "id": "2",
            "name": "SLi 2",
            "srcNode": "2",
            "dstNode": "4",
            "resources": [{
                "name": "bandwidthResource",
                "bandwidth": "100000000"
            }]
        },
        {
            "id": "4",
            "name": "SLi 4",
            "srcNode": "4",
            "dstNode": "6",
            "resources": [{
                "name": "bandwidthResource",
                "bandwidth": "1000000000"
            }]
        },
        {
            "id": "8",
            "name": "SLi 8",
            "srcNode": "6",
            "dstNode": "9",
            "resources": [{
                "name": "bandwidthResource",
                "bandwidth": "1000000000"
            }]
        },

    ]
    }}
```

3.5 EVALUATION

The evaluation considers the validation of the approaches using the fixed topology of the use case, and the runtime, acceptance ratio, and path utilization metrics for the chain embedding heuristic.

3.5.1 *Applying the Methods to the Use Case*

Our fixed topology that represents the use case includes 12 ARs, as shown in Figure 3.1. These ARs represent the communication between the factory hall level and edge computing level, and communication between the edge computing level and cloud computing level. The ARs in Figure 3.1 represent AR categories, but the real ARs in the definition files include specific end-points, for example, a certain type of sensors. We show in this section the generated EVN for a specific AR (AR11) in Figure 3.8. We also show the relevant part of our substrate topology defined in ALEVIN with the mapping results for AR11 in Figure 3.9.

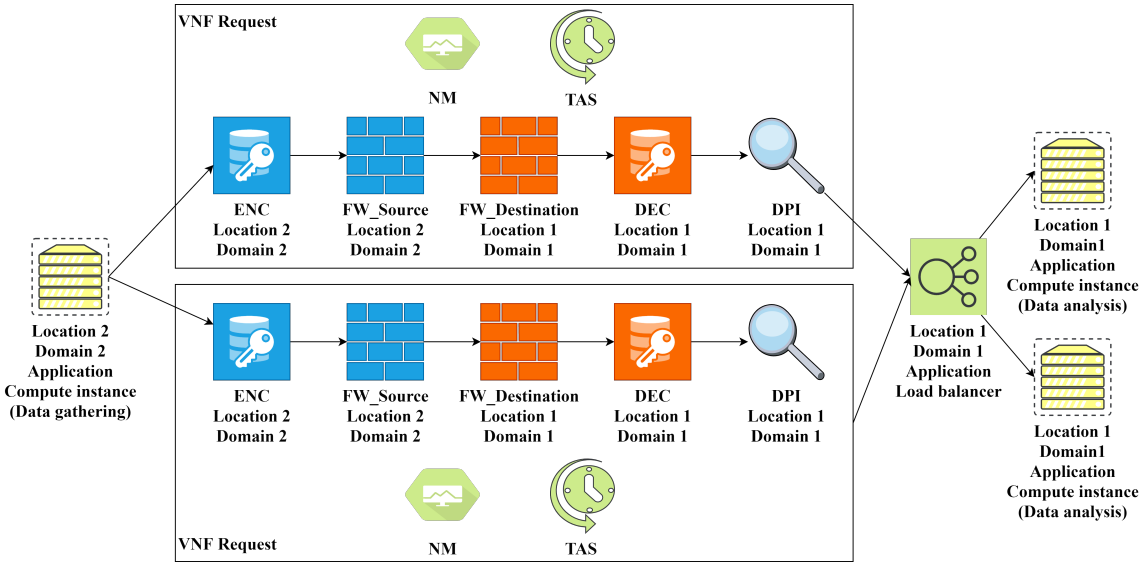

Figure 3.8: Generated EVN for AR11 [3]

AR11 connects an application VNo (data gathering) from the edge computing level (location 2, domain 2) to an application VNo (data analysis) from the cloud computing level (location 1, domain 1). AR11 defines redundancy and low latency requirements and connects different locations and domains. For these requirements, two redundant VNF requests are created with encryption, firewall, DPI, and monitoring VNFs. Furthermore, TAS VNF is added.

The resulting FGs from the chain composition stage are similar to Figure 3.5. As mentioned before, TAS VNF is only added to the beginning of the FG and the chain embedding algorithm adds it to each server that hosts the FG. A load balancer is added and the target data analysis VNo is cloned, according to the calculated CPU demand and server capacity. The mapping results in Figure 3.9 show the mapping of the two redundant FGs over two completely disjoint paths (dotted line for FG1 and dashed line for FG2).

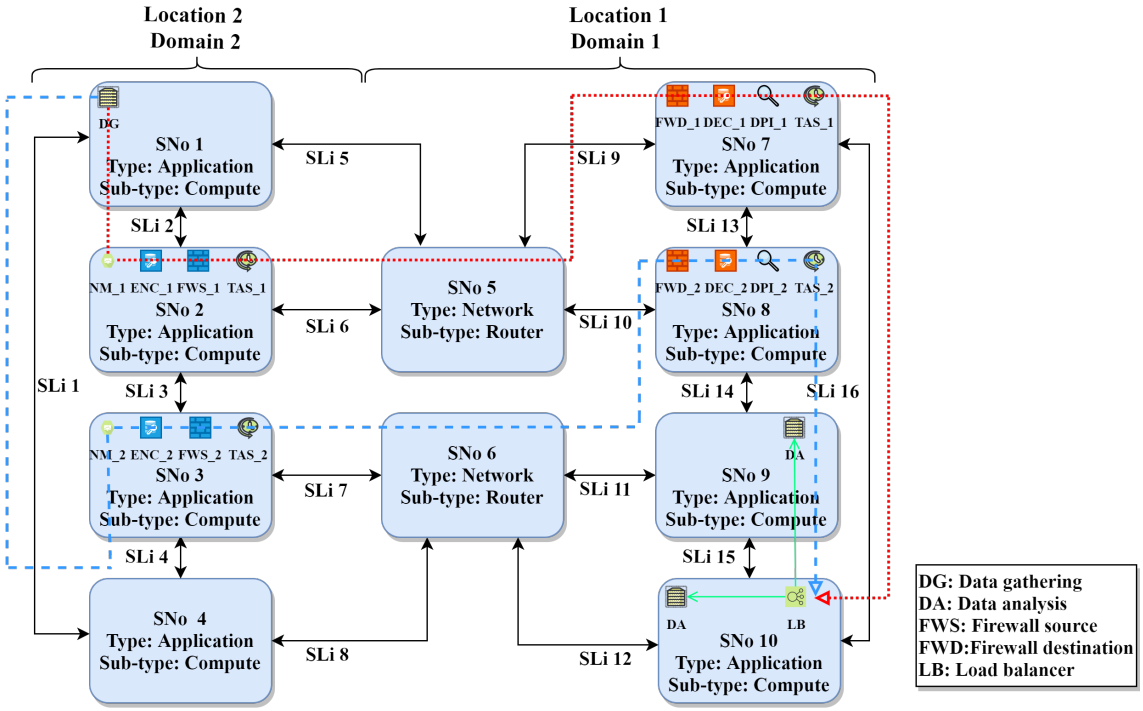

Figure 3.9: Embedding AR11 in the SN [3]♮

The mapping results are represented using a JSON format that shows for each VNo the mapping SNo and for each simple VLi the mapping path. However, for VLis with VNF demand, the node and link mappings of the FG are combined. For the FG internal VLis, the types of the source and destination VNFs are shown. The SNo that hosts a set of VNFs and relevant FG VLis are shown by SLis for which the source and destination are that SNo. Listing 3.3 depicts a partial embedding result for one SFC from AR11 that belongs to one of the VLis between the DG and LB applications functions.

Listing 3.3: Mapping result

```
{
  "nodeMapping": [
   {
         "ID": "1",
         "name": "DG",
         "MappingSNo": "1"
   },
   {
         "ID": "2",
         "name": "LB",
         "MappingSNo": "10"
   }
  ]
  "linkMapping": [
    {
      "ID": "1",
      "VNFDemand": "yes",
      "SrcSNo": "1",
```

```
19      "DstSNo": "10"
20      "mapping": [
21          {
22            "linkType": "SLi"
23            "SrcNode": "1",
24            "DstNode": "2"
25          },
26          {
27            "linkType": "SLi"
28            "SrcNode": "2",
29            "DstNode": "2"
30          },
31          {
32            "linkType": "FG"
33            "srcNetworkFunction": "NM",
34            "dstNetworkFunction": "ENC"
35          },
36          {
37            "linkType": "FG"
38            "srcNetworkFunction": "ENC",
39            "dstNetworkFunction": "FWS"
40          },
41          {
42            "linkType": "FG"
43            "srcNetworkFunction": "FWS",
44            "dstNetworkFunction": "TAS"
45          },
46          {
47            "linkType": "FG"
48            "srcNetworkFunction": "TAS",
49            "dstNetworkFunction": ""
50          },
51          {
52            "linkType": "SLi"
53            "sSrcNode": "2",
54            "sDstNode": "5"
55          },
56          {
57            "linkType": "SLi"
58            "sSrcNode": "5",
59            "sDstNode": "7"
60          },
61          {
62            "linkType": "SLi"
63            "sSrcNode": "7",
64            "sDstNode": "7"
65          },
66          {
67            "linkType": "FG"
68            "srcNetworkFunction": "",
69            "dstNetworkFunction": "FWD"
70          },
71          {
```

```
            "linkType": "FG"
            "srcNetworkFunction": "FWD",
            "dstNetworkFunction": "DEC"
          },
              {
                "linkType": "FG"
                "srcNetworkFunction": "DEC",
                "dstNetworkFunction": "DPI"
              },
              {
                "linkType": "FG"
                "srcNetworkFunction": "DPI",
                "dstNetworkFunction": "TAS"
              },
              {
                "linkType": "SLi"
                "sSrcNode": "7",
                "sDstNode": "10"
              }]
      }
    ]
}
```

With such a complex environments and set of constraints, evaluating the algorithms with random topologies and using traditional VNE metrics is challenging. The efficiency of the approach can be judged through the theoretical complexity. What is significant to both the provider and customer is the validation of the mapping results. In this perspective, we suggest using validation policies mapped to the original policies. The input of these policies shall be the SN topology, the initial AR, the final EVN, and the mapping results. The patterns are validated without rules or transformations and without considering intermediate EVNs. These policies can be described using a mix between an algorithmic language and our transformation patterns. Another main difference between the transformation and validation patterns is that there is no need to follow a pre-defined order when applying the validation patterns.

Algorithm 5 depicts an exemplary validation pattern to check if all and only all devices required by a certain AR are included in the EVN and its mapping results. This algorithm shall be called for each EVN. However, the validation is more complex than the transformations but less complex than chain composition and EVN embedding algorithm. Several similar validation policies/algorithms are required to check all policies.

3.5.2 *Runtime with Fixed SN*

In this evaluation scenario, we check the runtime of the whole approach with the previous fixed SN topology and two different representative ARs. In each step, we add another copy of this set of ARs. The runtime results are depicted in Figure 3.10, and show linear and low runtime with this fixed topology. This is expected from the complexity calculation' that shows that the runtime is linear with the EVN size.

Algorithm 5: Node type validation

```
1  ValidateNodeType ()
2      Input: SN(N_SN, E_SN), AR(N_AR, E_AR), EVN(N_EVN, E_EVN), EVN mapping results
3      for n ∈ N_AR do
4          if n.type = "device" then
5              for m ∈ N_SN do
6                  if m.type = "device" then
7                      if n.subtype = m.subtype AND n.domain = m.domain AND
                        n.location = m.location then
8                          if m ∉ N_EVN OR m ∉ nodeMapping(EVN) then
9                              return False
10                         end
11                     end
12                 end
13             end
14         end
15     end
16     for m ∈ N_SN do
17         if m.type = "device" then
18             for n ∈ N_AR do
19                 if n.type = "device" then
20                     if n.subtype ≠ m.subtype OR n.domain ≠ m.domain OR
                        n.location ≠ m.location then
21                         if m ∈ N_EVN OR m ∈ nodeMapping(EVN) then
22                             return False
23                         end
24                     end
25                 end
26             end
27         end
28     end
29     return True
30 end
```

3.5.3 *Evaluation of the Chain Embedding Using a Random Topology*

Our chain embedding algorithm is a form of greedy heuristics, since it adapts to the remaining path length and next path resources when making the placement decision. When the path is long, the algorithm tries to spread the VNFs over the path. When the path is short, the algorithm tries to place the VNFs on the least possible number of servers. We compare it to an existing similar solution to show its feasibility. Our VNF consolidation method, the random topology, and the distribution of resources are chosen to represent hierarchical edge computing where the resources might be limited and paths are short. Comparison to the results of other works might be challenging, since there are differences in the objectives and topologies.

The evaluation structure in ALEVIN enables the comparison of different algorithms using random topologies and several metrics. This structure also enables the definition of the number of runs and the assignment of random resource/demand values from specified ranges. The created SN and VNs, their demands and resources, mapping results,

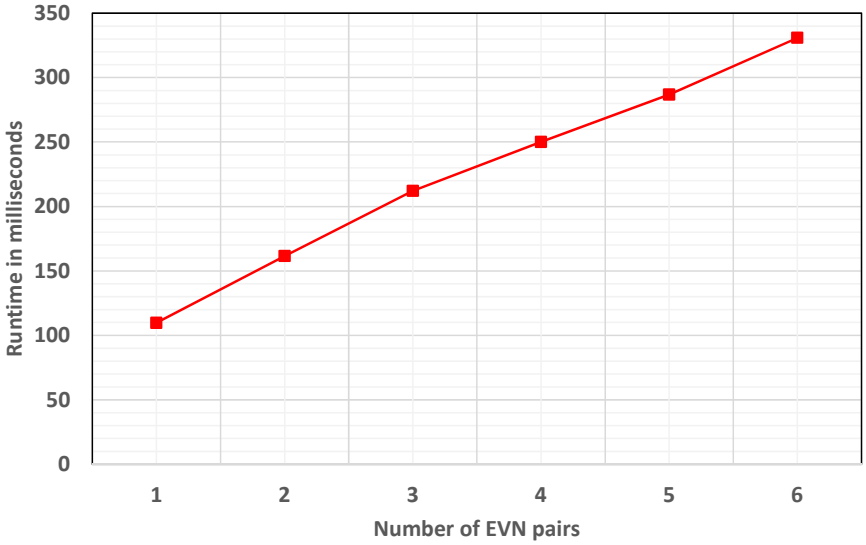

Figure 3.10: Runtime with varying number of EVN (pairs)

and resulting metrics' values are exported to an XML file that includes the results for all runs.

In the SFC embedding domain, two graph models are used for generating random networks. The Waxman model [149] used in [46], creates highly randomized networks by placing in each step two nodes on a two-dimensional plane and connecting them with a certain probability. This probability is calculated from their distance and two model parameters α and β, where the large β increases the edge density and small α increases the probability of shorter edges.

The Barabasi Albert model [150] used in [53] creates random scale-free networks by taking preferential attachment into account. With preferential attachment, the degree distribution follows a power law and the probability of connecting two nodes is based on individual nodes' degrees. Initially, a certain number of nodes (m_0), time steps, and edges to be added per time step are defined. Based on the defined number of time steps, a new node with $m \leq m_0$ edges is added to the network in each step. A new node is connected to an already existing node i that has a degree k_i with a probability:

$$P(i) = \frac{k_i}{\sum_{jk_j}}$$

Figure 3.11 is an exemplary generated topology with the parameters:

- Initial number of nodes: $m_0 = 1$
- Number of time steps: 20
- Number of new edges per time step: 1

ALEVIN uses directed graphs and the generator adds the reversed edge for each created edge. The Barabasi Albert model is better for representing industrial network levels (as in our use case). We use it for generating the SN for evaluating the chain

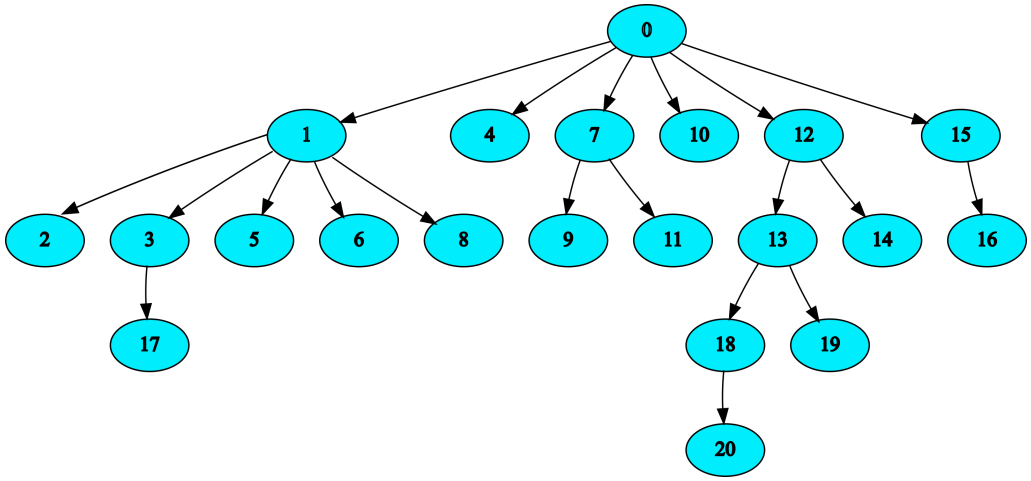

Figure 3.11: A random topology generated by Barabasi-Albert generator [3]

embedding algorithm. Each VN has two VNos and a VLi with a VNF demand, in which we vary the chain length. We adapted our EVN embedding algorithm for this evaluation by just performing node mapping for VNos based on ID demand and using the k-shortest path algorithm to select the path. Subsequently, the chain embedding heuristic is used to map the FG without the locations, domains, and types of nodes. Creating random SN that includes such properties to evaluate the whole system and choosing algorithms to compare with is a challenging future work.

Our system runs in an offline mode, where the SN and EVNs are defined in advance. Each test run is repeated 30 times and the mean values with confidence intervals (0.95%) are graphically depicted. We use two typical metrics from VNE; the runtime and acceptance ratio. The runtime is the time needed by the algorithm to embed a number of EVNs on a SN. The acceptance ratio is the ratio of successfully embedded EVNs to the total number of EVNs in the scenario. We define a new metric to represent the efficiency in utilizing SN resources for chain embedding. Average path utilization is the ratio of SNos that are utilized multiple times to embed the FGs. The metric value is 0 if each VNF from a FG is mapped to a different substrate node:

$$\text{Average Path Utilization} = \frac{\text{Number of multi-utilized SNos}}{\text{Number of VNFs in all FGs}}$$

3.5.3.1 *Runtime*

Runtime evaluation is performed for an increased SN size and increased number of EVNs to be embedded. The resource/demand values for these scenarios are adjusted, such that there is no rejection of EVNs. We compare the runtime of the chain embedding to a scenario with the same parameters in which there is no VNF demand and only bandwidth is verified over the path.

RUNTIME FOR INCREASING SN SIZE In this scenario, the number of EVNs is fixed to 50, where each FG contains four to eight VNFs. The SN size is increased in each simulation to judge how much time it takes to embed a fixed number of EVNs with increasing SN size. The results are presented in Figure 3.12 and they show a small overhead of

the chain embedding, on average 250 ms for medium-size SN of 300 SNos. However, the runtime grows with a linearithmic trend. This matches the total complexity of the solution $O(N.M + L(E + N.logN))$. linearithmic time complexity $O(nlogn)$ is slightly worse than a linear complexity $O(n)$, but much better than a quadratic complexity $O(n^2)$.

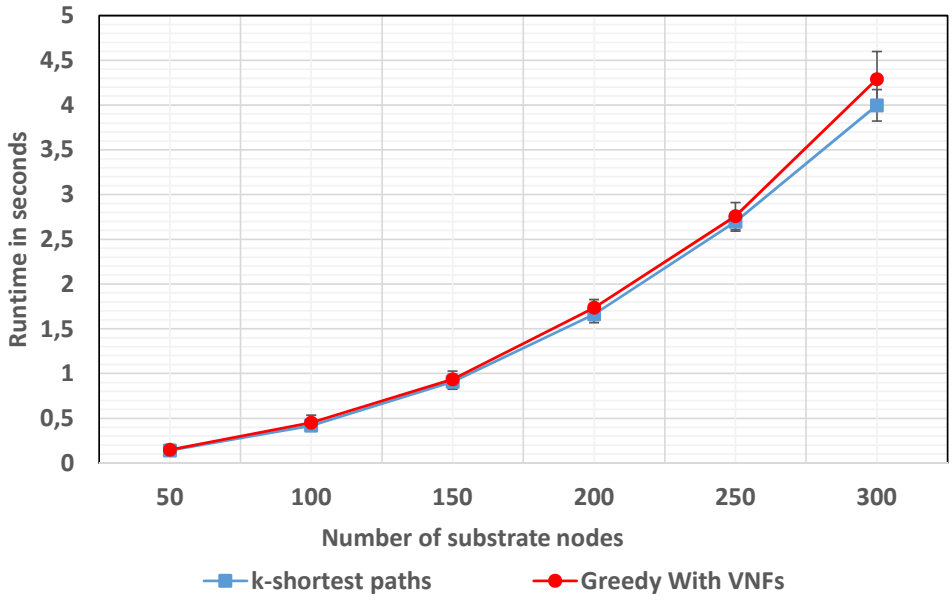

Figure 3.12: Runtime for increased SN size [3]‡

RUNTIME FOR INCREASED NUMBER OF EVNS In this experiment, the SN size is fixed to 200 SNos and the number of EVNs to be embedded is varied. The number of VNFs in each FG is a random number between four and eight. Figure 3.13 depicts the results for this evaluation. For fixed SN size, the runtime is linear and there is no overhead of chain embedding for this size.

3.5.3.2 *Comparison to LightChain*

A simplified version of LightChain from [47] is implemented to compare our greedy chain embedding algorithm to another algorithm. LightChain algorithm tries to allocate the FG on the shortest path between two end-points. If the chosen shortest path is consumed, then the algorithm calculates another shortest path and the remaining VNFs are allocated. LightChain approach does not allow for placing multiple VNFs on a single SNo. Our algorithm can place multiple VNFs that might belong to different SFCs on a single substrate node as long as enough capacity is available. If the selected path cannot be used, then another shortest path is tried. The behavior of the algorithm reduces the probability of path rejection.

ACCEPTANCE RATIO WITH INCREASING FG LENGTH In this scenario, we evaluate how the acceptance ratio changes when the number of nodes in the FG increases. The SN size is fixed to 100 nodes and 100 EVNs are to be embedded. The FG length varies in

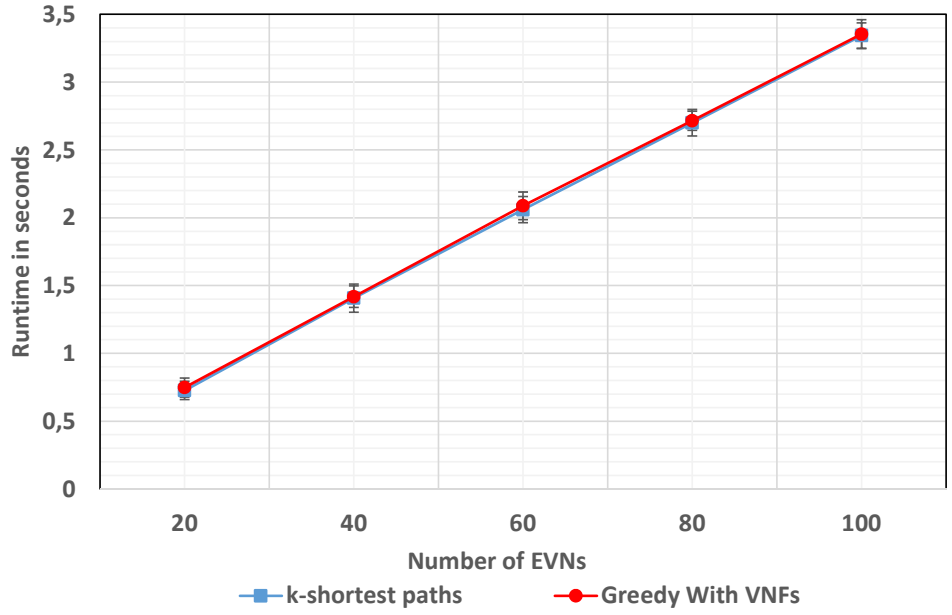

Figure 3.13: Runtime for increased number of EVNs [3]‡

the range 1–12. The resources and demands are highly relaxed, such that rejection is not because of resources, but due to the VNF mapping strategy. The results in Figure 3.14 show that the acceptance ratio of the LightChain solution highly decreases with increasing FG length and fixed SN size. These rejections are met when the paths are shorter than the FG. Our chain embedding algorithm keeps a 100% acceptance ratio since it is adaptive to FG length.

AVERAGE PATH UTILIZATION WITH INCREASING FG LENGTH With the same experiment parameters, Figure 3.15 shows the average path utilization with increasing FG length. The value for the LightChain approach is constantly zero, since the embedding strategy does not allow to place multiple VNFs of a FG on the same SNo. This is different from our greedy approach, where the utilization of SNos increases for longer chains.

ACCEPTANCE RATIO FOR INCREASING CPU CAPACITY Comparing the acceptance ratio for scenarios where resources are limited is also important. Therefore, in this scenario, we measure the acceptance ratio with increasing CPU resources, while the bandwidth is highly relaxed. The SN includes 100 SNos and the number of EVNs is 100. Based on the results in Figure 3.14, when each FG has five nodes, both algorithms can reach a high acceptance ratio when the resources are highly relaxed. The CPU demand of each VNF is fixed to one. As depicted in Figure 3.16, the acceptance ratio for our chain embedding approach is rapidly increasing and it reaches 100% acceptance for a CPU capacity of 15. For the LightChain approach, the acceptance ratio is also increasing, but at a lower rate. The results clearly depict that occupying multiple SNos on a path results in a much higher acceptance ratio.

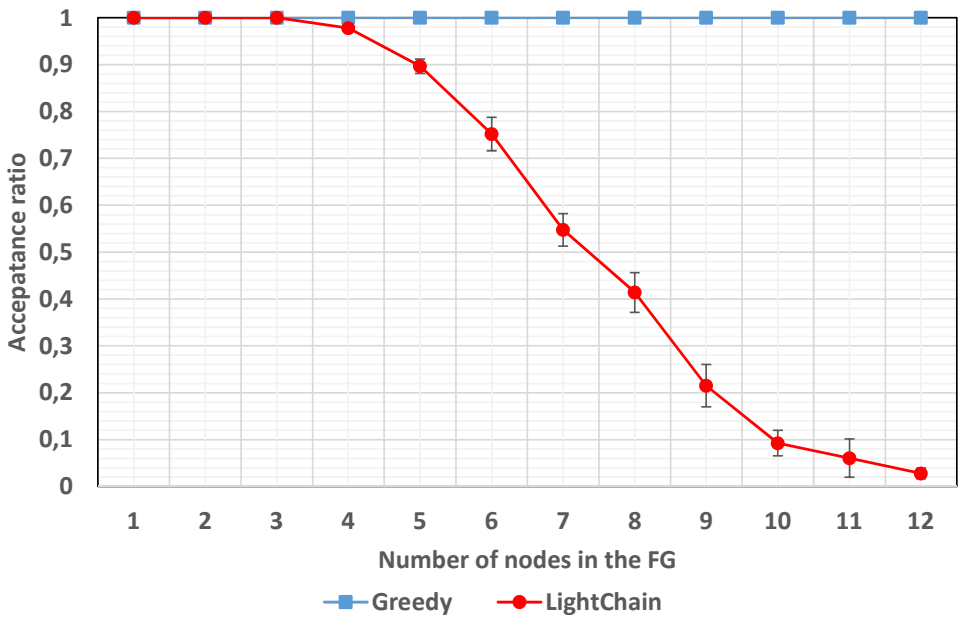

Figure 3.14: Acceptance ratio for increased FG size [3]♮

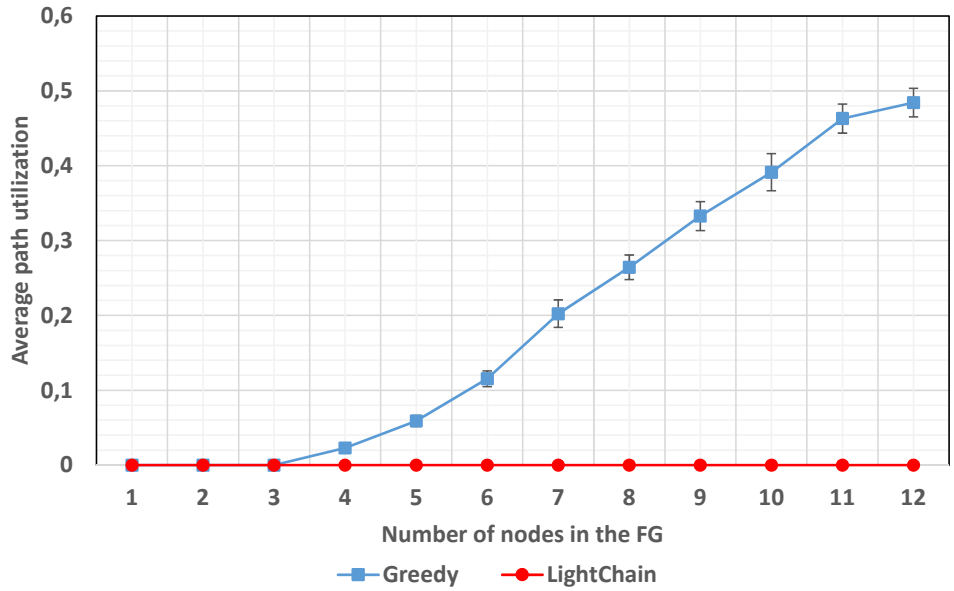

Figure 3.15: Average path utilization for increased FG size [3]♮

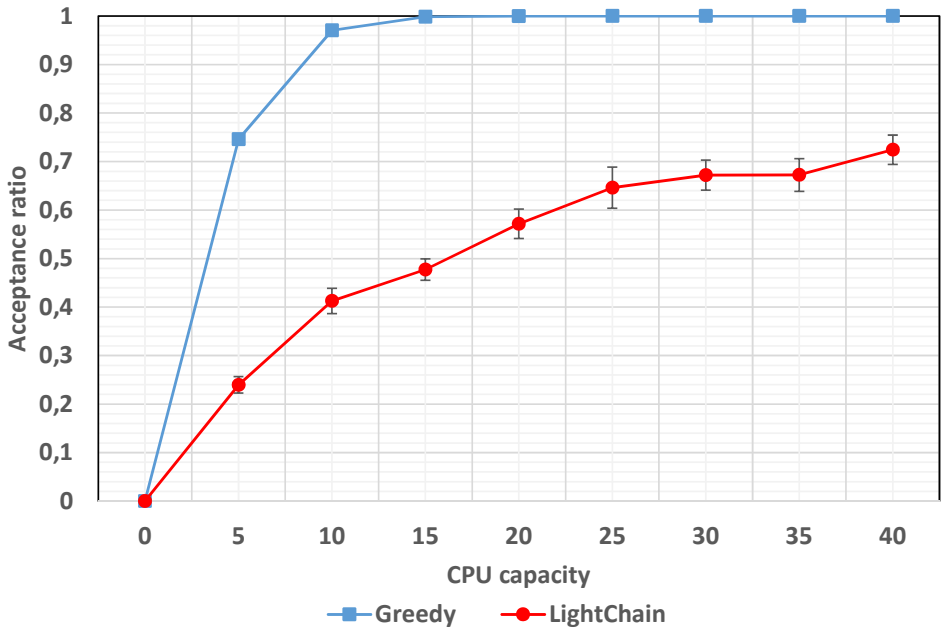

Figure 3.16: Acceptance ratio for increased CPU capacity and FG length of 5 [3]‡

3.6 CONCLUSION

Remote asset management is important for future industries and requires flexible and reliable network services that can be provided by means of network virtualization. However, the underlying network infrastructure imposes several performance, security, and resilience challenges. To address these challenges, NFV that supports flexible composition and deployment of SFCs can be used. However, these NFV procedures shall be automated for large industries based on the application requirements and network service provider policies, such as the location of VNFs and required security mechanisms.

This chapter presents a model that flexibly creates and deploys EVNs and SFCs based on these requirements and policies. The model applies a set of rules sequentially on an AR using graph transformation. The resulting graph is an EVN that includes the application nodes and the composed SFCs based on its requirements. The chapter also proposes a topology-aware heuristic to embed the SFC based on path early verification. We implement the developed methods and our use case in ALEVIN and present an exemplary mapping result. This result shows that our system can correctly compose and map the EVN and SFCs while satisfying all policies.

An evaluation of the chain embedding heuristic using a typical random topology, shows that it is promising for such environments in terms of admission, resource utilization, and performance. A main challenge for future work is evaluating the entire approach using random topologies that hold the required properties of entities. Furthermore, determining and implementing comparable solutions from the existing works is required. Another challenge is more coordination of the stages by SFC recomposition according to the embedding results. Finally, the future work will focus on completely defining

the validation policies that check the correctness of the whole EVN composition and embedding approach.

4

PERFORMANCE PERSPECTIVE - VIRTUAL TSN

This Chapter is an extension of the author's publication [2][‡].

In this chapter, we discover the feasible methods of using NFV to reduce the complexity and cost of implementing and deploying the TAS. TAS is one of the traffic shapers presented in the recent IEEE TSN standard for industrial Ethernet [151]. Virtualizing network functions decouples them from proprietary hardware but imposes performance challenges. We research high performance NFV techniques and use DPDK to implement a virtual TAS with a feasible performance for enterprise-level industrial applications.

Furthermore, we design a complete framework that provides schedule calculation, transmission selection, TAS controller, and time synchronization. Additionally, the framework includes evaluation tools; traffic generation and performance measurement. We evaluate our virtual TAS using delay and frame loss metrics, and a small SFC. The evaluation considers different loads of BET and external disturbance, and varied TCT specifications.

4.1 INTRODUCTION

NFV decouples network functions from physical devices using standard virtualization techniques. A network function becomes a piece of software running inside a VM that is easy to configure and deploy in different locations. This mechanism provides the flexibility to chain the functions to build the required network services dynamically. Furthermore, network functions running inside the VMs are easy to upgrade according to new standards (or their updates) and to adapt to changes in the requirements without broad investment in equipment. However, virtualization can degrade the performance of network functions by randomly delaying the processing. Additionally, the standard hardware is not optimized for network functions.

TSN [151] is a set of new standards from IEEE that enhance the Ethernet allowing it, among several aspects, to handle TCT together with the lower priority traffic (BET) on the same network. A traffic shaper from the standard, TAS, achieves this through pre-planning of the exact times of arrival and transmission of TCT in each port, and reserving the required time slots in which BET is blocked. Theoretically, TAS guarantees zero queuing delay and very low jitter for TCT (deterministic latency). However, to achieve that, an exact schedule calculation is needed, global time synchronization is mandatory, and no randomness in the processing time of the frames is allowed.

The concept of smart factory in Industry 4.0 is based on data analysis and autonomic decision making in real-time. For example, data can be collected from sensors and leveraged to control customizable production, safety, energy consumption, and security. These applications might impose different classes of traffic (in terms of time-criticality) that share the same network, making TSN a feasible technology to realize them. However, such scenarios require high network reliability and availability, and acceptable energy consumption. These objectives can be supported by edge computing and virtualization

technologies, for example, by deploying real-time network services using NFV. This can be achieved by leveraging specific capabilities of virtualization technologies, such as auto-scaling, live migration, and latency-aware SFC composition and deployment.

The combination of both technologies, TSN and NFV, applies the flexibility and scalability of NFV to TSN. Furthermore, this will ease the adaptation to the changing needs without broad investment in equipment, as well as upgrading the internal functioning of TSN to the future changes in the standard. However, TSN requires highly performing hardware with deterministic behavior, and NFV introduces processing overhead that might be stochastic. Nevertheless, some enhancements exist in the area of high performance NFV, such as DPDK that bypasses the network stack of the OS (see Section 4.4.4). A motivation to virtualize TSN is that the target smart factory applications might have less strict latency requirements than the manufacturing control applications.

In this chapter, we research the mechanisms that can be used to implement virtual TAS using high performance NFV. Then we design, implement, and evaluate a virtual TAS as an SFC composed of multiples TAS-capable VNFs. Furthermore, we develop a complete framework with an NFV-specific TAS controller, scheduling and transmission selection algorithms, time synchronization, and traffic generation and performance measurement tools. We propose a new method to support the schedule calculation, mainly in virtual environments, by measuring the real transmission and processing times in advance (prefetching).

In the evaluation, we measure the packet loss and delay of TCT traversing a TAS SFC. We use different scenarios to judge the capability of virtual TAS in providing comparable performance to the hardware-based TAS designed in the standard. Additionally, we evaluate the effect of BET traversing the same SFC, and external disturbance traversing a secondary SFC that uses the same server. These factors are significant in analyzing the virtualization overhead.

4.2 HIGH PERFORMANCE VNF

The deployment of a high performance VNF is possible using two different technologies, hypervisor/VM based and container-based. The container-based approach is particularly questionable with security issues. For an industrial enterprise that transfers data to the edge or cloud, data security is critical. On the other hand, the hypervisor/VM approach is the basis on which the architecture of NFV and its standards have been developed.

The main advantage of the containers is that they are light weighted. Containers require less resources than VMs and impose less overhead since they run the guest applications directly on top of a host OS while keeping them isolated. VMs run their guest applications on top of a guest OS and need a hypervisor to provide isolation and management. Besides eliminating the guest OS, a container does not need a hypervisor to manage the VNFs. Instead, by means of OS-level virtualization, containers can present the same functionalities as VMs in a very light manner. The container-based approach supports the NFV deployment in terms of scalability thanks to the easiness of adding new virtual functions and dividing the VNFs into smaller entities that have certain degree of independence (microservices). Furthermore, sharing the same OS between different containers removes all the hypervisor overhead in packet processing and the guest OS and allows Direct Memory Access (DMA) techniques, which speed up the communication among the VNFs themselves and with the external network elements.

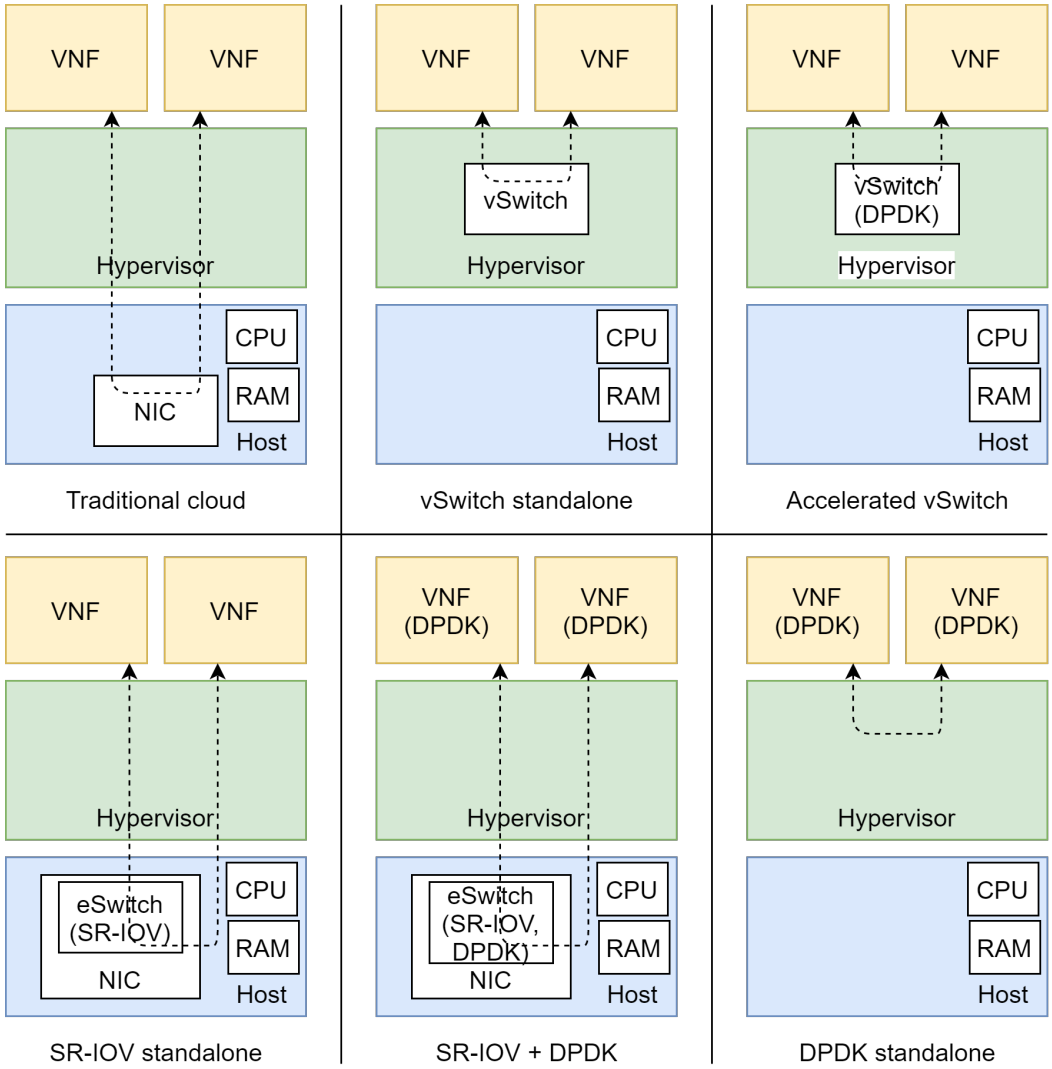

Figure 4.1: Existing high performance NFV mechanisms

The main deficiency of the container-based approach is the intolerable multi-tenancy security threats resulting from sharing the host OS.

The standard NFV architecture from the European Telecommunications Standards Institute (ETSI) [152] presents a VNF layer that uses the virtual resources offered by the NFVI. High performance VNF can be achieved by increasing the number of VNFs or their components but unless obliged, this method is not recommended [153]. A better method is to enhance all the layers from the hardware to the applications running on the VM. We focus on the possible improvements on the application level and referring to the changes required in the other layers for the enhancements to take effect. Figure 4.1 summarizes the primary high performance NFV approaches existing in the literature. In the following, we summarize and compare these approaches.

DPDK STANDALONE This approach integrates data processing acceleration libraries into the network stack of the VNF or relies on protocols that use them and use drivers compatible with the operating CPU. One tool discussed in [61] and [154] and [155], is DPDK [57]. This set of libraries and drivers works with a different logic than the

traditional network stack that waits for interruption and reacts to them (interrupt-driven mode). DPDK polls the data coming to the Network Interface Card (NIC) (poll-mode) directly to the user space where it is processed. Consequently, it avoids the kernel space and all the imposed overhead. Two conditions are to be verified before using this approach. The NIC should support poll mode, and the processor should be compatible with DPDK.

ACCELERATED VSWITCH This approach does not apply any changes to the application itself rather it changes the user domain in the hypervisor, namely the vSwitch. A vSwitch is a software running in the hypervisor to enable virtual networking for the VMs. Each VM has its own vSwitch's socket that allows it to communicate with the other VMs via the hypervisor without using the NIC. For this approach, the vSwitch will be assisted by data processing acceleration libraries (e.g., DPDK) in forwarding the packets between the application and the physical NIC and between the VMs. With the accelerated vSwitch, all the networking (VM/NIC or north-south traffic and VM/VM or east-west traffic) is conducted in the user-space. One example of vSwitch is the Open vSwitch.

SINGLE ROOT I/O VIRTUALIZATION (SR-IOV) This approach treats the performance problem from the hardware (NIC) side by enhancing the computing capabilities of the hardware to allow application direct access and bypass the hypervisor. It also offloads the packet processing from the CPU to the NIC, contrary to previous approaches that treat the problem from the software (VNF or vSwitch) side and reduce the CPU usage by migrating the packet processing to the user domain. Enabling SR-IOV in the NIC card allows it to provide multiple virtual copies of the PCI function, named Virtual Functions (VFs). Every VF can be attached to a VM, allocating it a direct access to the physical resource. With such a method, multiple VMs can use the same NIC without the need for the hypervisor since the virtualization is provided by the NIC itself. Consequently, All VMs are offered a line-rate networking performance and have direct and full control of the network resource part assigned to them.

COMBINED APPROACH Another interesting approach is to combine SR-IOV with DPDK without using a virtual switching since most of the functions offered by virtual switches can be executed by the eSwitchs coming with the SR-IOV solution. The authors of [61] applied this technique in implementing Deep Packet inspection (DPI) of network traffic in the form of a VNF. The authors showed a very good performance compared to the traditional cloud architecture. Unfortunately, they did not compare their approach to the SR-IOV standalone approach or the accelerated vSwitch approach.

APPROACHES COMPARISON In a comparative study performed by INTEL [61], the DPDK-accelerated vSwitch approach and the SR-IOV standalone approach were compared under the same exact setup with two different traffic flow patterns (north-south pattern and east-west pattern). The results showed that for the north-south flow pattern, SR-IOV is a better choice than DPDK-accelerated vSwitch. However, for the case of east-west flow pattern, DPDK-accelerated vSwitch performed better than SR-IOV. The reason behind these results is the nature of the traffic itself:

- North-south traffic is the traffic between the server on which the VM is deployed and the external network. Thus, every transmission or reception of a frame should travel

from the VM to the NIC. In the DPDK-accelerated vSwitch solution, each frame should pass through the hypervisor. Although the procedure is accelerated, the CPU processes all the frames while SR-IOV establishes a direct mapping between the VM and the NIC.

- East-west traffic is the traffic within the same server between the different VMs. If SR-IOV is used, each frame is forwarded to the NIC to be processed and forwarded back to the destination VM. In this case, the vSwitch is faster since the NIC is not needed and the whole operation can be performed in the user domain.

We choose to combine two of the suggested architectures to build DPDK-accelerated VNFs running on top of a DPDK-accelerated virtual switch. Our implementation has been evaluated on a single server. Thus, the communication is inside the same hypervisor domain and there is no need for a physical NIC.

4.3 ASSUMPTIONS AND SIMPLIFICATIONS

TSN standard [151] specifies the implementation details, such as data models and functions. In our work, we adopt several simplifications and assumptions and adapt the implementation to our tools and context. However, we implement the main functionality of TAS in traffic scheduling, and the required tools for performance evaluation. According to the standard, the main TSN traffic scheduling capabilities are: supporting multiple traffic classes, enhancements for scheduled traffic (gating mechanism), state machines for scheduled traffic (gating mechanism), and frame preemption. A possible extension is to implement per-stream filtering and policing.

From another perspective, and according to the standard, the forwarding process inside a bridge is composed of eight stages: topology enforcement; ingress filtering; frame filtering; egress filtering; flow metering; frame queuing; queue management; and transmission selection. In an outbound queue, First-In-First-Out (FIFO) transmission is used, and one traffic class is queued. The standard defines a range of eight traffic classes, and this can be adopted partially or totally by an implementation. The queuing algorithm decides where to enqueue a frame based on its internal priority value. The transmission selection method uses three state machines.

The first state machine watches the time and initiates the GCL execution that is performed by the second state machine. The third state machine is for the GCL configuration. Since the focus of this work is on handling scheduled traffic through gating mechanism and performance measurement in the context of NFV, we work on the last three stages: queuing frames, queue management, and transmission selection. However, we perform metering at the traffic class level in our controller.

In this work, we implement two stages, enqueuing/dequeuing, for inserting/removing the frames in/from the queues, respectively. The queue management is simplified into removing each frame after one sending trial, regardless of the result. It is also combined with the transmission selection. Our transmission selection algorithm merges the first two state machines in a "select and watch" method. Our algorithm reads the GCL entry and keeps watching the clock until its respective execution time. Then it applies the entry, reads the next, and starts selecting frames for transmission while watching the clock. We implement the GCL configuration state machine in the control plane, and it is executed

only once before the start of each experiment. Furthermore, we define one Rx-queue and one Tx-queue per port and one FIFO queue per traffic class.

Another simplification in this work is the flow routing and deployment of the VNFs. We pre-configure the routes in Open vSwitch, which interconnects the pre-deployed VMs that execute TAS. Thus, no routing information is required. Only VLAN tags are used to identify the types of frames (TCT, BET, control frame and its type).

The last simplification is that our schedule calculation algorithms consider one path for all flows, do not try to compress/minimize the schedule length, and do not use delay constraints in the schedule calculation. However, the advanced schedule calculation methods that try to minimize the length of the GCL are more important for hardware than software that has high flexibility. Currently, we don't have an estimation of frame processing time, and we schedule each frame based on its expected transmission time using experimental data.

4.4 FRAMEWORK DESIGN

Our solution has two main independent but complementary components. The first component is the control and measurement unit that plays the role of the Precision Time Protocol (PTP) master node [156] for synchronizing the clocks of VNFs. This unit also generates and distributes the TCT schedule, generates TCT traffic based on the schedule, and records statistics on the received frames (talker and listener roles). The second component is the TAS-capable virtual bridge or VNF that applies the pre-configured schedule to forward the frames. It also synchronizes its clock with the master clock before starting the forwarding. For evaluation purposes, a third component is a BET generator that produces randomly-sized frames with a random Inter-Arrival Time (IAT).

4.4.1 *Control and Measurement Unit*

The controller runs three main processes, as depicted by Fig. 4.2. The control process performs the synchronization and scheduling using the information gathered by a prefetch message per TCT flow. The second process is the frame generation process. This process generates and transmits frames for all scheduled flows according to the pre-calculated schedule. An identifier of the frame is included to recognize the frames by the reception process. The exact sending time of each frame and the count of frames sent during the experiment are also recorded. The third process is the frame reception process that receives frames, identifies them, and records their reception times as well as the count of the successfully received frames per traffic class. In the synchronization and schedule distribution operations, two modes are used; centralized or chained.

In the centralized mode, the synchronization and schedule configuration for each VNF is performed in a separate round by the controller. In the chained mode, each VNF in the chain is configured by the previous VNF. The centralized and chained SFC configuration (TCT deployment) are inspired by the TSN Centralized Network Configuration (CNC) and TSN distributed configuration, respectively. Chained deployment is vital for handling several long chains.

In the network prefetch operation, the controller sends a prefetch frame through the SFC. Each VNF forwards this frame after inserting its exact reception and transmission

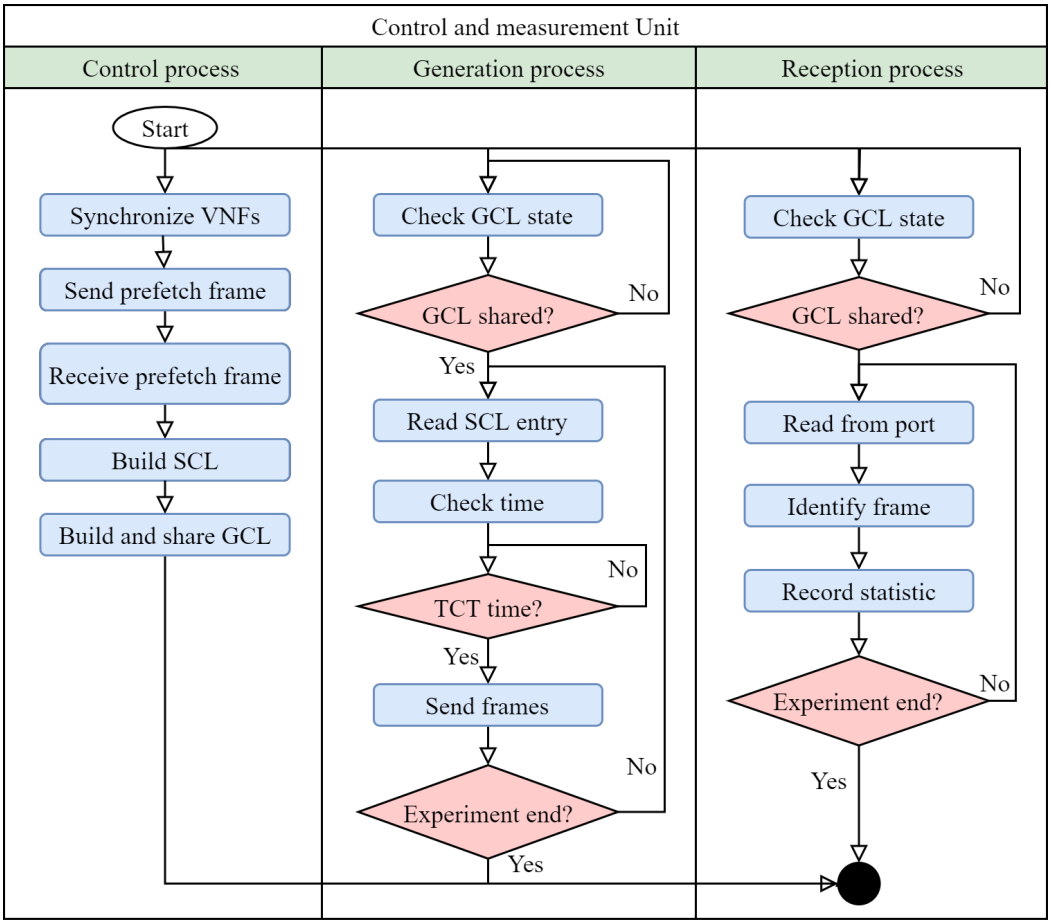

Figure 4.2: Controller flowchart [2]

times. After receiving back the frame, the controller uses this timing information from the SFC for schedule calculation. In the schedule calculation operation, GCL is constructed in two steps. The first step is the calculation of the Sending Control List (SCL) that includes the talker sending times for all flows. The second step is the adaption of the SCL to the VNFs, which results in a different copy of the GCL for each. This list is dynamic, and it depends on the way we estimate the start and the end times of the transmission operation of each frame in each VNF. Theoretically, the start of transmission time shall be synchronized with the TAS gate opening event, and the end of transmission shall be synchronized with the TAS gate closing event. In our work, we use two algorithms that are adapted to our context. Both algorithms use the SCL in building the GCL. However, the first algorithm is empirical, and it is based only on the information gathered by the prefetch phase. The second algorithm is hybrid, and it has limited use of the prefetch data (see Section 4.5.2).

4.4.2 *TAS-capable VNF*

As shown in Fig. 4.3, the TAS-capable VNF separates the data plane and the control plane, and has separate queues for TCT, BET, and control frames. The traffic frames are inserted into the reception ring (Rx-ring) until the enqueue thread sorts them. This thread identifies the priority class of incoming frames and inserts the TCT and BET frames in

their respective queues. The dequeue process retrieves and sends the frames based on the GCL entries and the synchronized clock.

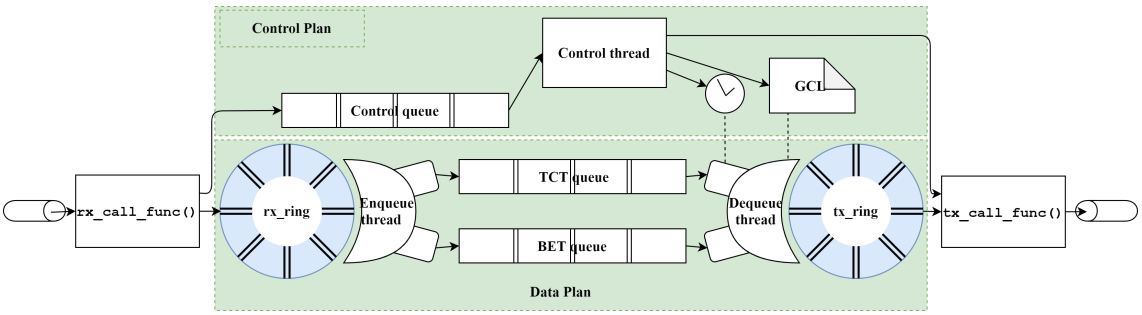

Figure 4.3: Architecture of the TAS-capable VNF [2]‡

The control thread establishes the GCL and the globally synchronized clock required for the dequeuing operation. We have five types of VNF control frames and respective operations on them:

- Prefetch frame: the frame is forwarded after inserting its reception and transmission time.
- PTP frames: the required/received time-stamps are inserted/recorded, or the clock is synchronized.
- GCL frames: a GCL entry that includes gate opening and closing times is added.

4.4.3 *BET Generator*

The BET has two parameters, generation rate (number of frames per second) and frame size. The frame generation follows the Poisson distribution pattern [157]. The choice of Poisson distribution is conventional when characterizing traffic within a data center [158]. The IAT between two frames in a Poisson distribution follows an exponential distribution:

$$IAT = \frac{-1}{\lambda} * \ln(1-u)$$

Where λ is the Poisson rate, and u is a random number. Generating random frame sizes using Poisson distribution can be done using the formula:

$$Size = \sum_{x=0}^{x=u} \frac{e^{-\lambda} * \lambda^{-x}}{x!}$$

4.4.4 *Tools*

DPDK [57] is a set of libraries written in C programming language. It offers a complete framework for fast packet processing in the data plane. It uses the poll mode that allows the DPDK based application to avoid the overhead of the traditional interruption-based mode. DPDK controls the NIC and buffers the incoming frames using Direct Memory Access (DMA) in the user space, which bypasses the network stack of the OS. The throughput is highly improved for DPDK-based applications, but the NIC cannot be

shared with other applications. DPDK treats packets in bursts and can balance between throughput and latency. DPDK uses huge memory pages, and the resources needed by an application must be reserved Before launching it. DPDK offers several functions to handle the memory, packets, queues (rings), and timing (Table 4.1).

Table 4.1: Main DPDK functions used in our implementation

Function	Role
rte_rdtsc()	Returns the value of the time stamp counter
rte_eth_rx_burst ()	Reads a burst of packets from an input queue
rte_eth_tx_burst()	Writes a burst of packets to an output queue
rte_ring_create ()	Creates a packet ring (i.e. TCT-queue)
rte_pktmbuf_free()	Frees the storage of a packet into the mempool
rte_ring_sp_enqueue ()	Appends a packet to a ring
rte_ring_sc_dequeue_burst()	Retrieves several packets from a ring
rte_eth_add_rx_callback()	Defines a callback function in a port
rte_pktmbuf_pool_create()	Creates memory pool for packet storage
rte_eal_mp_remote_launch()	Launches a function on a logical core

The second tool we used is Open vSwitch [159]. It is an open-source multilayer virtual switch that can be used to interconnect VMs over multiple physical servers across the network. It is usually combined with a Hypervisor to offer massive networking. The configuration of the vSwitch simplifies the routing of frames and setting up the evaluation chain without using MAC addressing. This configuration of the vSwitch includes the following steps:

- Setting up vSwitch to use DPDK acceleration.
- Constructing a bridge.
- Adding sockets to the bridge and attributing them to the specific VNFs to use them.
- Setting up the flows among the sockets.

Finally, the Kernel-based Virtual Machine (KVM) hypervisor is used to host the VNFs. KVM allows for high-performance and low latency [160], and is the most widely adopted compute hypervisor in the OpenStack community [161]. OpenStack is a typical NFVI supported by ETSI. From another perspective, container-based virtualization is lightweight but imposes security challenges to industrial networks, since it does not provide a similar isolation level to hypervisors.

4.5 SCHEDULING ALGORITHMS

The scheduling logic first calculates the SCL used for flow transmission at the talker, then it calculates the GCL for each VNF using two different methods and the prefetching

information. Finally, the runtime logic is applying the SCL at the talker and the GCLs at the VNFs to select the frames for transmission.

4.5.1 *SCL Calculation Algorithm*

The SCL includes the periods of sending TCT frames by the talker (controller generation process). These periods are based on the cycles of flows and are repeated in an SCL cycle that is the least common multiple of the flows' cycles. Each flow might have multiple sending periods inside the SCL cycle. Algorithm 6 checks the next sending time of all flows and inserts the soonest in the SCL while prioritizing the flows with the smallest cycles. Even if the flows have equal cycles and frame sizes, they are scheduled sequentially. We also assume that a set that includes one frame from each flow can be transmitted during the smallest flow cycle. This assumption and the scheduling method prevent the overlapping of flows, and the same applies to the GCL calculation. The TCT specifications in the evaluation scenarios are chosen accordingly.

In Algorithm 6: N^i is the number of sending periods of flow i in the SCL cycle; *LCM* is the least common multiple; *TTs* is the list of transmission times of all flows in the SCL cycle; *TT* is the transmission time of a flow in an SCL entry; *T* is the current time; and *FTP* is one flow transmission period in an SCL entry. Figure 4.4 shows an exemplary SCL calculation for three flows with different cycle times (periods). Only a portion of the SCL is shown with the numbers of frames refer to their order in the calculated SCL. We notice for time-overlapping entries (such as 12 and 13), that the flow with the smaller cycle is prioritized.

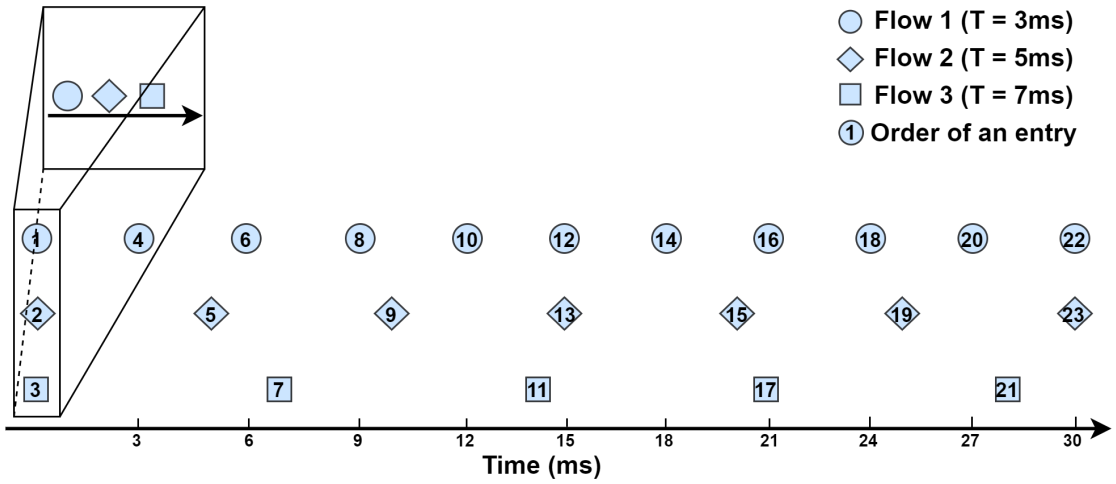

Figure 4.4: Example of SCL calculation

4.5.2 *GCL Calculation Algorithms*

The empirical algorithm is based only on the information gathered in the prefetch phase (Fig. 4.5). The prefetch method is a form of calibration of the virtualization overhead, randomness of processing delay in servers, and the fluctuation of network bandwidth at the beginning of each experiment. This calibration is performed using one prefetch

Algorithm 6: SCL calculation

```
1  CalculateSCL ()
2      //Initialization
3      SCL.cycle = LCM(All flow^i.cycle)
4      for flow^i in flows do
5          N^i = SCL.cycle / flow^i.cycle
6          for k = 0 To N^i do
7              //Entries of transmission times: (flow ID, flow cycle, transmission
                time)
8              TTs.append(flow^i.ID, flow^i.cycle, k * flow^i.cycle)
9          end
10     end
11     //Ordering the list of transmisson times based on the times then flow
        cycle for equal entries
12     AscendingOrder(TTs, TT, cycle)
13     for entry^j in TTs do
14         //SCL entry: (Transmisson time, flow ID)
15         //Adding a flow sending time to the SCL
16         SCL.append(max(SCL.T, entry^j.TT), entry^j.FlowID)
17         //Updating the current time point in the SCL based on the flow's frame
            transmission period (frame size/bandwidth)
18         SCL.T = SCL.T + entry^j.flowID.FTP
19     end
20     //Updating the SCL cycle to be the reached scedule length
21     SCL.cycle = SCL.T
22 end
```

frame per flow that is treated as TCT. An assumption here is that the loads on the servers used for TAS and on the network are not highly fluctuating such that this calibration is valid during the experiment. Based on this, if an SCL entry m is related to flow i, the respective GCL entry in the VNF^k of the SFC is:

$$VNF^k.GCL[m].OT = SCL[m].TT + (Tx_i^k - SendT_i)$$

$$VNF^k.GCL[m].CT = VNF^k.GCL[m].OT + (Rx_i^{k+1} - Tx_i^k)$$

Where OT/CT are the TCT gate opening/closing times, Tx_i^k/Rx_i^k are the transmission/reception times of the prefetch message of flow i in VNF^k, and $SendT_i$ is the transmission time of the prefetch message of flow i in the controller. This GCL entry represents the TAS gate opening/closing events that match the respective SCL entry in a certain VNF. The added value $(Tx_i^k - SendT_i)$ represents the time required for the flow-specific prefetch message to start its transmission by VNF^k, which represents the previous transmission and processing delays in the SFC. The closing event depends on the time required until the next VNF receives the prefetch frame $(Rx_i^{k+1} - Tx_i^k)$.

The hybrid algorithm mixes the theoretical calculation with prefetching data that are only used to estimate the processing time of a frame in a VNF. The algorithm calculates the transmission period of each frame based on its size and the link bandwidth. Then, it combines the prefetching information to calculate the GCL entries. Consequently:

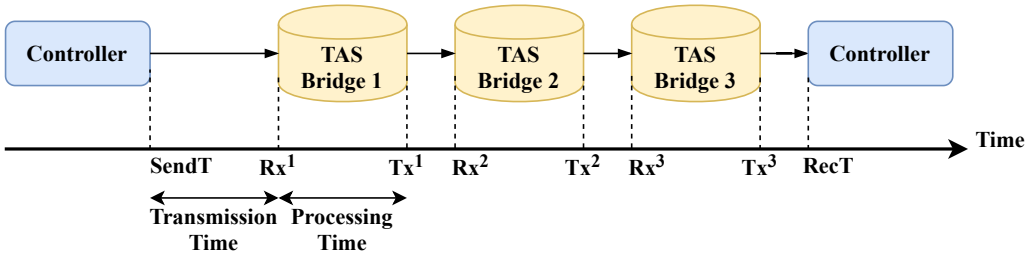

Figure 4.5: Prefetch information for one flow [2]‡

$$VNF^k.GCL[m].OT = SCL[m].TT+$$
$$\sum_{j=1}^{k} \frac{SCL[m].FlowID.FS}{VNF^j.BW} + \sum_{j=1}^{k}(Tx_i^j - Rx_i^j)$$

$$VNF^k.GCL[m].CT = VNF^k.GCL[m].OT+$$
$$\frac{SCL[m].FlowID.FS}{VNF^{k+1}.BW}$$

where FS is the frame size of a flow, and $VNF^j.BW$ is the bandwidth of the incoming link to VNF^j. In the OT equation, right side, the second and third forms represent the frame transmission and processing periods in the SFC, respectively. For both algorithms, the current time value is also used for the GCL calculation to avoid the overlapping of flows:

$$If\ (OT < GCL.T)\ then\ OT = GCL.T$$
$$GCL.T = CT$$

4.5.3 *VNF Transmission Selection Algorithm*

The transmission selection (dequeuing) process has a default behavior of forwarding BET while waiting for the schedule (GCL). Once the GCL is received, the entries are processed sequentially, and this processing is repeated after each scheduling cycle. For each entry, queued TCT frames are transmitted in bursts until the gate closing time. Then the queued BET frames are transmitted in bursts before the next gate opening time. These operations are shown in Algorithm 7, where BS is the burst size, Ts is the schedule cycle start time, SC is the scheduling cycle, TP is the total experiment period.

4.6 EVALUATION

Our evaluation scenarios are designed based on the following objectives:

- Evaluating the performance of the virtual TAS for TCT with different traffic specifications.
- Comparing our two schedule calculation algorithms.
- Measuring the mutual influence between TCT and BET traversing the same SFC.

- Measuring the effect of external disturbance.

The evaluation SFC, as depicted by Fig. 4.6, contains three TAS-capable virtual bridges (VNFs) besides the controller and a BET generator. All the links are configured in the Open vSwitch to be bidirectional except the link from the BET generator to the first TAS bridge, which injects the non-scheduled BET in one direction. The transmission of TCT is unidirectional starting from controller port 1, entering the SFC through port 0 of Bridge 1, and received at controller port 0.

Algorithm 7: Transmission selection

```
1  Transmit ()
2      while GCL not received do
3          frames ← dequeue(BET_queue, BS)
4          send(frames)
5      end
6      // Apply schedule
7      Ts ← T0
8      // Handling BET before any gate opening time
9      while ref_clock() < Ts + GCL[0].OT do
10         frames ← dequeue(BET_queue, BS)
11         send(frames)
12     end
13     while ref_clock() < TP do
14         for i = 0 To GCL.Size do
15             // Handling queued TCT frames before the gate closing time
16             while ref_clock() < Ts + GCL[i].CT do
17                 frames ← dequeue(TCT_queue, BS)
18                 send(frames)
19             end
20             // Handling BET before the next gate opening time
21             while ref_clock() < Ts + GCL[i + 1].OT do
22                 frames ← dequeue(BET_queue, BS)
23                 send(frames)
24             end
25         end
26         Ts ← Ts + SC
27     end
28 end
```

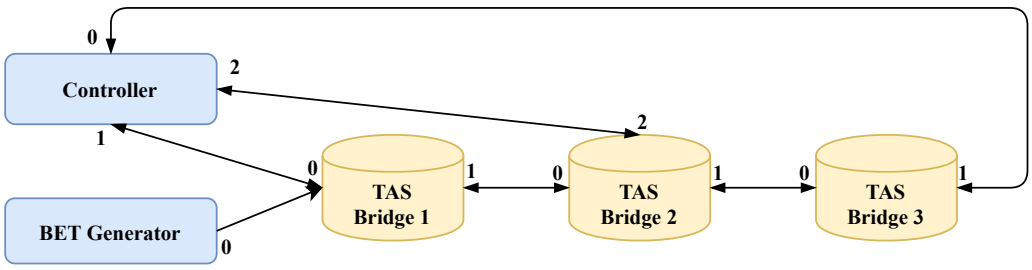

Figure 4.6: Evaluation SFC [2]

For scenarios 2 and 3, we use three TCT flows with specific cycles and frame sizes, as shown in Table 4.2. However, the third scenario considers different TCT specifications.

The cycle times are chosen to be prime numbers to increase the length of the scheduling cycles and represent the worst scenarios.

Table 4.2: Specifications of TCT flows

Flow	Cycle (ms)	Frame size (bytes)
flow 0	3	128
flow 1	5	512
flow 2	7	1024

The effect of BET is evaluated by varying λ; the rate of BET flowing in the main SFC. The results are gathered in the controller node by recording the sending and reception times of each frame. In our experiments, we send at least 10000 TCT frames. For the delay values, we count the frames in a certain delay range and confidence intervals are not applicable.

Our evaluation has been performed on a single server. We estimate that using multiple servers that are either directly connected or through TAS hardware, will not significantly change the results since using DPDK allows direct NIC access. We note here that using virtual TAS in real environments requires either using only standard servers as network hardware or TAS-capable network hardware with the deployment of a schedule synchronized with the TAS SFC schedule. Furthermore, we emulate the network disturbance in our evaluation, and our empirical scheduling algorithm considers bandwidth fluctuation. We use a burst size of one for the transmission selection, such that each frame can be transmitted in time to achieve low latency.

4.6.1 *Scenario 1 - Deployment Time*

In the first scenario, we evaluate the SFC configuration time or TCT deployment time. This is the time needed for synchronizing the bridges, the prefetch phase, and calculating and distributing the schedule. This is the period between the experiment's start and the beginning of forwarding TCT frames. In this scenario, we compare the centralized deployment performed through direct virtual links between the controller and TAS bridges, and the chained deployment performed across the SFC. We record the average deployment time for both modes over ten experiments. The results depicted in Table 4.3 show better performance with the centralized deployment due to the time needed for the TAS bridges to configure the next bridges sequentially.

Table 4.3: Deployment time results

	Average time (ms)	Confidence interval
Chained deployment	9022.4	0.52
Centralized deployment	6025.6	2.95

4.6.2 *Scenario 2 - TCT Delay and the Effect of BET*

In this scenario, we compare the empirical and hybrid scheduling algorithms in terms of frame delay for TCT. Furthermore, we evaluate the effect of the BET rate on this delay. In Fig. 4.7, we compare the cumulative percentage of frames per delay value (Cumulative Distribution Function (CDF)) for TCT when using the empirical and hybrid scheduling algorithms and $\lambda = 600$. The results show that 97% of the frames have a delay value less than 89ms when the hybrid algorithm is used, and less than 73ms when the empirical algorithm is used. The performance of the empirical algorithm is better since it depends on the prefetch information for both the transmission and processing time, while the hybrid algorithm uses the theoretical bandwidth. However, we have noticed high jitter in the delay, which requires further investigation.

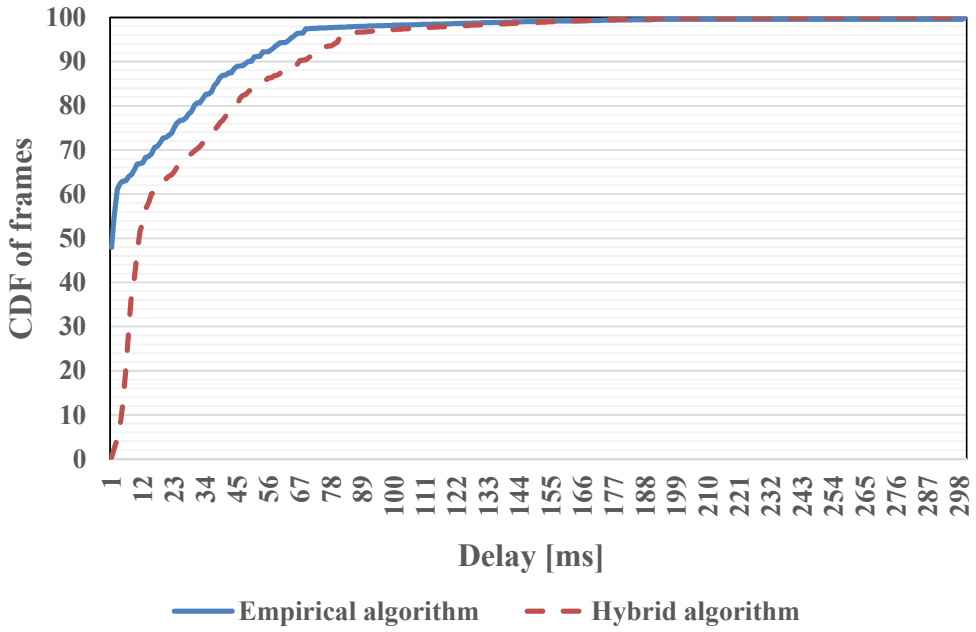

Figure 4.7: TCT delay CDF with different scheduling algorithms [2]‡

In the following experiments, we use the less performing scheduling algorithm (hybrid) to check the effects of BET and external disturbance. In Fig. 4.8, we compare the TCT delay CDF for different values of the BET rate (λ). The results show that this effect is limited but random due to the virtualization (processing) overhead.

4.6.3 *Scenario 3 - External Disturbance*

In this scenario, we evaluate the robustness of the virtual TAS by deploying a secondary SFC composed, as depicted by the figure 4.9, of two VNFs: a noise traffic generator and a traffic relay. The first node generates traffic of rate μ and forwards it to the second node that forwards it back to the source. The primary and secondary SFCs share the Open vSwitch but not the links and the ports. We compare the delay and frame loss with different disturbance levels in the secondary SFC. These levels are zero external

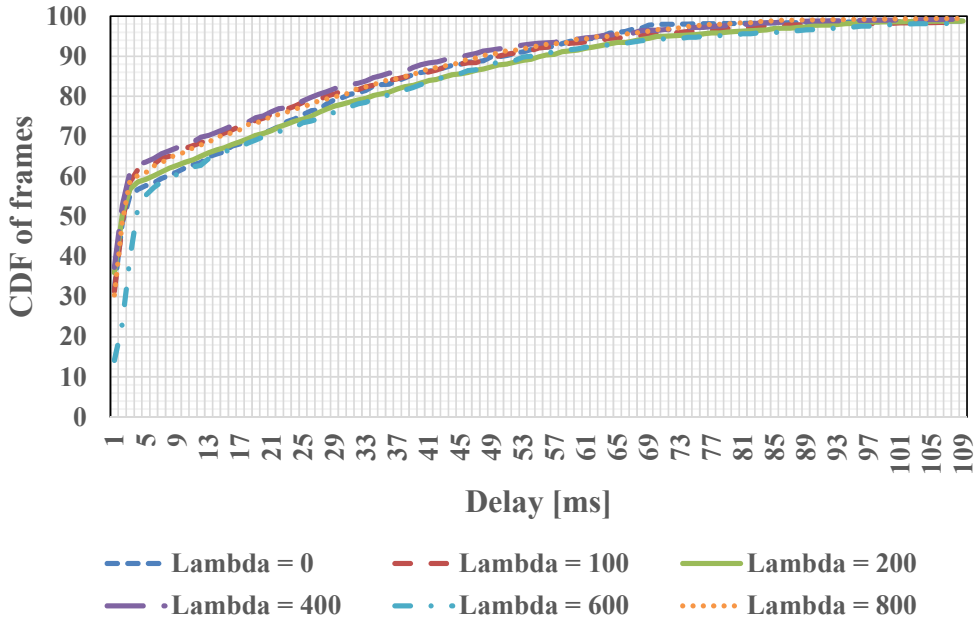

Figure 4.8: TCT delay CDF with varying BET rate [2]

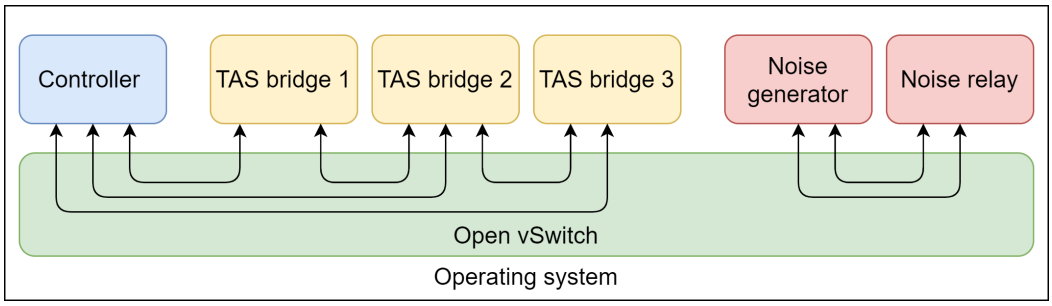

Figure 4.9: Disturbance chain topology

disturbance $\mu = 0f/s$ (frames per second), Mega-disturbance $\mu = 1600f/s$ and Giga-disturbance $\mu = 1600000f/s$, with an average frame size of 750 Bytes.

FRAME LOSS Lost frames are the frames that are not received during the experiment time. In Figures 4.10 and 4.11, we show the frame loss percentage for TCT and BET with the three disturbance levels, when varying the BET rate λ. For TCT, when $\lambda \leq 600$, the BET and disturbance cause small increasing effects. When $\lambda = 800$, frame loss is high due to the processing overhead on the VNFs, and the impact of disturbance is random. For the BET loss, its rate has an approximately linear effect, mainly with external disturbance. However, the exact causes of the frame loss need further investigation.

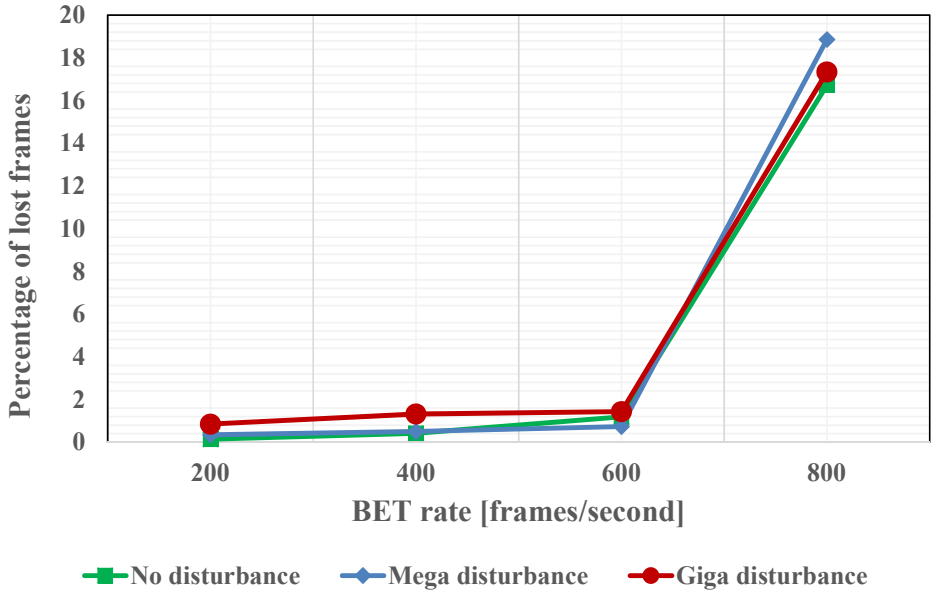

Figure 4.10: TCT frame loss ratio under external disturbance [2]‡

DELAY In Fig. 4.12, we show the delay CDF for TCT, $\lambda = 400$, and the three disturbance levels. 95% of the frames have delays of 76ms, 92ms, 154ms for zero, Mega, and Giga disturbance, respectively. These results show that the external disturbance has a significant effect on the delay due to the processing overhead. However, this effect does not grow fast with the growth of disturbance. When the disturbance is 1000 times higher (Mega to Giga), the delay range of 95% of frames increases with 62ms. These observations lead to the conclusion that a server should be dedicated to one TAS SFC that hosts TCT flows and a low rate of BET (less than 200 frames per second).

4.6.4 *Scenario 4 - TCT Specifications*

In this scenario, we perform four experiments with different numbers of flows and various specifications, as shown in Table 4.4. We evaluate TCT delay with $\lambda = 400$. The results are presented in Fig. 4.13 and show that adding one flow with a cycle of 1ms and a small frame size improves the delay (about 100% of frames with delay $\leq$ 73ms). This

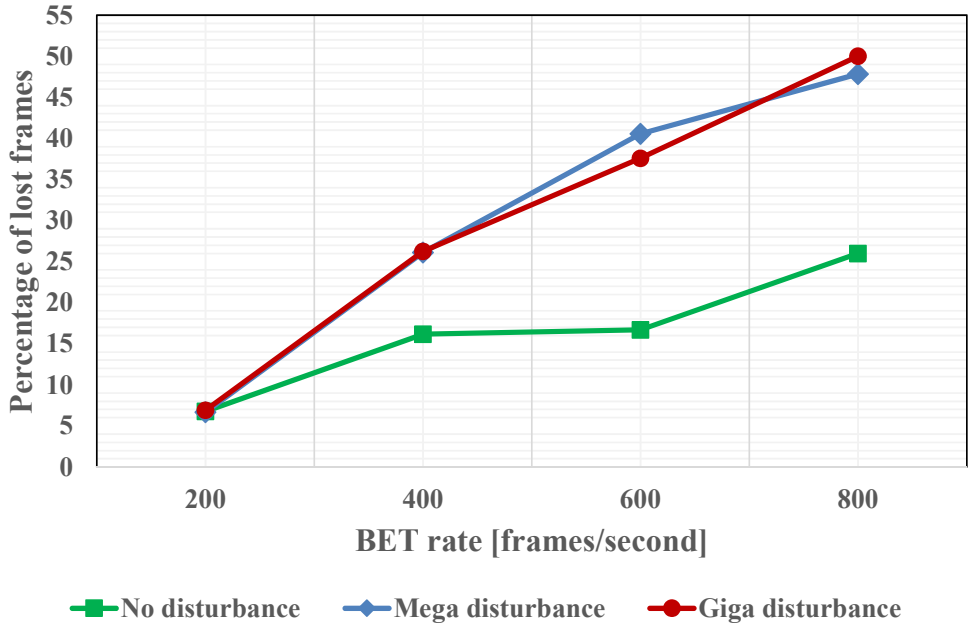

Figure 4.11: BET frame loss ratio under external disturbance [2]‡

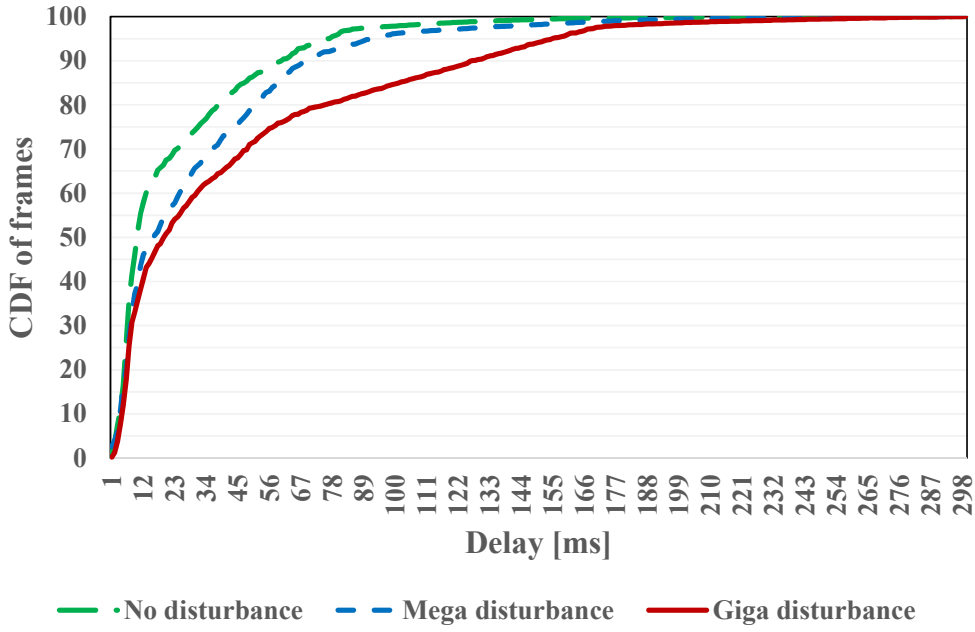

Figure 4.12: TCT delay CDF for varying disturbance and for $\lambda = 400$ [2]‡

result is due to more frequent TAS gate opening events. For experiments 3 and 4, we add more flows with larger cycles and frame sizes, but reduce the frame size of some flows with small cycles (5 and 7 ms) by half. These traffic specifications lead to lower traffic density, and the delay is significantly improved (about 100% of frames with delay $\leq$ 10ms). However, the high frequency of the TAS gate opening increases the frame loss of BET, which needs further evaluation.

Table 4.4: TCT specifications for scenario 4

Experiment	Flow	Cycle (ms)	Frame size (Bytes)
Exp 1	flow 0	3	128
(3 Flows)	flow 1	5	512
	flow 2	7	1024
Exp 2	flow 0	1	64
(4 Flows)	flow 1	3	128
	flow 2	5	512
	flow 3	7	1024
Exp 3	flow 0	1	64
(5 Flows)	flow 1	3	128
	flow 2	5	256
	flow 3	7	512
	flow 4	11	1024
Exp 4	flow 0	1	64
(6 Flows)	flow 1	3	128
	flow 2	5	256
	flow 3	7	512
	flow 4	9	768
	flow 5	11	1024

The results mentioned above show that with a typical TCT density, high frequency of gate opening, and a small SFC, our virtual TAS can guarantee a delay of 10ms with a high probability. This delay value is large compared to the theoretical end-to-end delay in the standard. The standard assumes that the scheduled traffic is only delayed by transmission time plus a small processing time in the bridges. For example, a frame of size 512 bytes transmitted over four hobs (controller and 3 VNFs), and using Gigabit Ethernet, has a total theoretical delay of 31 microseconds, assuming that the processing delay in TAS hardware switches is 5 microseconds. The positive aspect is that this virtual shaper has very high flexibility, acceptable stability with disturbance, and can provide guarantees that are feasible for factory level applications but not control applications.

4.7 CONCLUSION

Implementing and deploying TAS using high performance NFV provides high flexibility and lower effort and cost. The current performance and deployment times are feasible for factory level applications under certain constraints, mainly TCT density and gate opening frequency. The evaluation showed that calculating a TCT schedule based on the

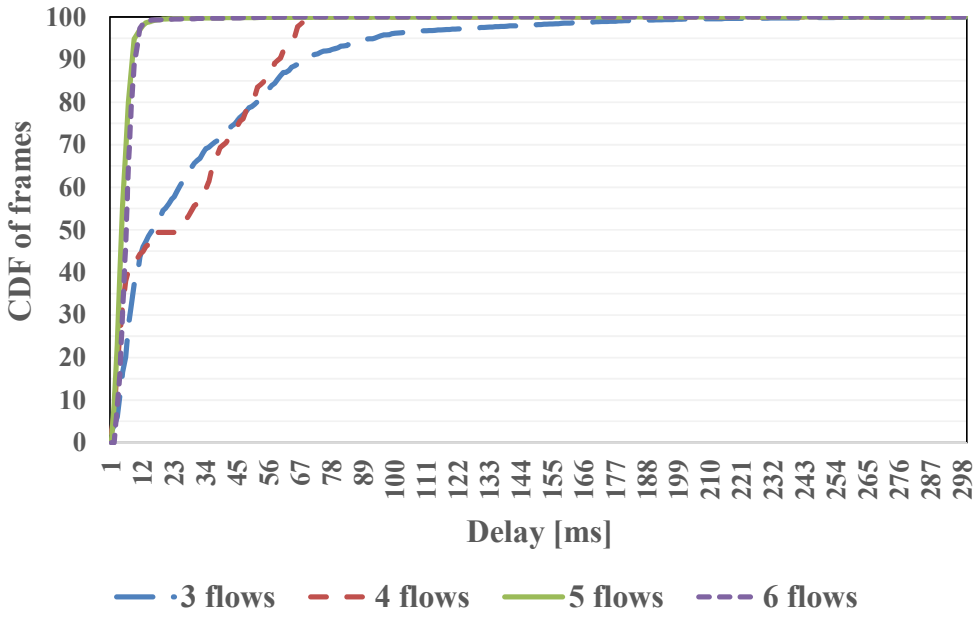

Figure 4.13: TCT delay CDF with $\lambda = 400$ and different specifications [2]‡

calibration of the environment improves the performance. Our virtual TAS is robust, for TCT, against medium BET traversing the same SFC, and medium external disturbance traversing a different SFC that uses the same computational resources. However, high BET load has a significant effect on the TCT frame loss, and high external disturbance has a significant impact on the TCT delay. This is due to the processing overhead resulting from virtualization, which also results in limited randomness in the delay. Several improvements to this work are significant:

- Implementing and comparing different scheduling algorithms that can consider nested SFCs and different paths.
- Comparing the performance to the TSN hardware.
- Making the clock synchronization and schedule dynamic by adjusting them during the experiment, to better control the environmental fluctuations.
- Investigating the virtualization overhead and its random effect on the delay (jitter) and frame loss, and researching techniques to control it.
- Implementing frame preemption and guardband methods.
- Implementing an interface of the framework to add/remove TCT flows dynamically.
- Using longer SFCs deployed over multiple servers.
- Flow identification and metering (per-flow statistics).
- Evaluating the effect of DPDK burst size.

5

RELIABILITY PERSPECTIVE - BRANCHING AND FRER

Link and node failures in the physical network are inevitable, at some point of time some entity might fail. In particular, having VNs on top of the physical infrastructure makes the failures very costly since one single failure can affect multiple paths of different VNs. In fact, different VLi mapping algorithms shall be used depending on the requirements of certain traffic classes. For traffic that requires high reliability, algorithms that maximize the reliability shall be used. For traffic that requires medium reliability, algorithms that satisfy reliability with reasonable resource utilization shall be used. For traffic that requires low reliability, traditional reliability-aware link mapping algorithms shall be used.

In this chapter, two algorithms for redundant path calculation are presented. In the first algorithm, Minimal Branching (MinBr), in case the reliability demand cannot be achieved by a single path, the smallest possible branching that satisfies this demand is computed. The second algorithm, Maximal Branching (MaxBr), searches for the longest path branching making the redundant paths as disjoint as possible and, in that way, adding more reliability to mapped path. Calculation of a backup path makes the VN resilient to any number of failures in one of the resulting branches. These two algorithms are compared to the Reliability Shortest Path (RSP) algorithm that calculates shortest paths using reliability as a weight, and therefore, improves the path reliability in comparison to reliability-agnostic solutions. Moreover, the path calculation method of this algorithm is used also by the two redundant path algorithms.

Our findings show that using redundancy algorithms increases the overall acceptance ratio of the Virtual Network Requests (VNRs) with high reliability demand. The MinBr mapping algorithm, with the least branching calculation, utilizes the resources in a very efficient way while satisfying the required reliability. While the longest branching algorithm, MaxBr, improves path resilience without increasing the rejections since fully disjoint paths are not mandatory.

From another perspective, integrating theoretical graph theory-based resilient VNE solutions with existing technologies is very important to make these solutions applicable in real networks to configure the redundancy policies in order to satisfy more complex requirements. We present in this chapter the integration of our resilient VNE solutions with the standard IEEE Std. 802.1CB FRER. Having four main functions (sequence encode/decode, sequencing, stream splitting, individual recovery), the FRER mechanism can increase the overall reliability of the physical infrastructure. Furthermore, we use an FRER-capable TSN simulator to integrate and test our algorithms implemented in ALEVIN. In ALEVIN, we compute network configurations compatible with this simulator.

5.1 INTRODUCTION

While providing high reliability to the applications, it is equally important not to congest the network with unnecessary replicated traffic and cause needless delays and inefficient use of resources. To avoid these issues, the traffic class shall define the level of required

redundancy. For non-deterministic traffic, we propose to use traditional link mapping algorithms with reliability as a weight and without redundant path calculation. Redundant paths shall be calculated for deterministic traffic. For cyclic traffic, we propose to always calculate a backup path. Whereas for the acyclic traffic, we propose to calculate a backup path only if a single path cannot fulfill the requirements of the VNR. Thus, for deterministic traffic, FRER capabilities of TSN shall be used.

We tackle the problem of efficient stream path calculation by means of network virtualization. There are different mapping algorithms in the field of VNE, which try to solve the problem of resilient VNs, such as online network re-configuration, and tree redundancy algorithms. Many of the proposed solutions take an assumption that the link and node failures occur randomly, and thus, they map a VLi on a path that might be prone to frequent failures. In this chapter, we introduce algorithms that take into consideration the reliability of each network component and redundant paths calculation when needed.

Based on the mapping results, the required configurations for FRER are computed. FRER provides two options to configure the network: manually creating all configuration tables, or auto-configuration which limits the flexibility of FRER to some extent. In this work, we automate the generation of all configuration tables while keeping the flexibility of FRER. Moreover, for validating this work, a TSN simulator is used and adapted to import and apply the generated configurations.

The goals of the chapter can be summarized as follows:

- Investigating branching techniques in order to provide more resilience to VNs.
- Evaluating these algorithms with a typical factory topology.
- Calculating network configurations in accordance with the IEEE Std. 802.1CB FRER based on the mapping results of the algorithms used.
- Validating the calculated network configurations using a network simulator.

The remainder of the chapter is organized as follows: Section 5.2 provides detailed background on FRER. Section 5.3 describes the different link mapping algorithms. Section 5.4 describes the methods of computation of network configurations. The implementation tools and details are described in Section 5.5. The evaluation scenarios and their results are discussed in Section 5.6.

5.2 FRER STANDARD

The focus of this work is on the TSN sub-standard 802.1CB FRER [162], which is specifically designed for the purpose of increasing network reliability by reducing the frame loss rates. The general idea is replicating frames of a stream at the source end system and/or in relay systems in the network, and eliminating those replicates in the destination end system and/or in relay systems. Additional objectives of the standard are the flexible deployment of its functions, latent error detection, robustness, dynamic capabilities, in-order delivery, and ease of use. In order to fulfill these goals, four main functions are defined:

- *Sequencing function* with two sub-functions: sequence generation function that generates the sequence number, and sequence recovery function that uses the

sequence number to decide which frame to pass and which to discard, operating on a set of member streams.

- *Stream splitting function* replicates frames of the input stream, assigning each replicated stream a different ID. It is allowed to have at most one replicated stream with the same ID as the original stream.
- *Individual recovery function* inspects the sequence number of an arriving frame. If it is a duplicate of an already received frame, the frame is discarded. This function operates on a single stream.
- *Sequence encode/decode function* encodes the generated sequence number into the frame by e.g. a Redundancy Tag (R-TAG), and extracts it and possibly removes the encapsulation from the frame.

Additionally, FRER requires *stream identification function* described by IEEE Std. 802.1AC. It is responsible for generating a stream ID from a received frame from lower layers, passing it up the stack. This function modifies the frame if enforced by the encapsulation method.

In this section, only a brief summary of the functions is presented. The above-mentioned functions as well as the respective configuration tables are further explained in subsections 5.2.3 and 5.2.4. The standard defines recommended and required functions of different parts of the network. Thus, not all functions have to be implemented in all network entities.

5.2.1 *System Types, Requirements and Recommendations*

As shown in Figure 5.1, the standard differentiates five types of systems in a network: talker end systems (originating compound streams), relay systems (forwarding or discarding packets of compound streams), listener end systems (consuming compound streams), stream identification components, and FRER C-components. In this work, the focus is on the talker, relay, and listener systems. The standard defines requirements and recommendations for each of them.

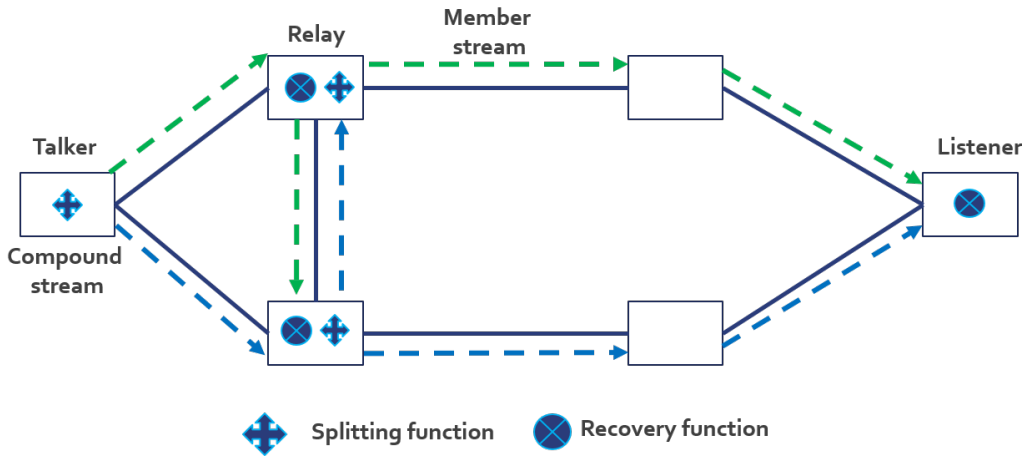

Figure 5.1: System types in FRER, adapted from IEEE 802.1CB ([162])

The requirements of the talker end system are the following out-facing functions on at least one port, and for at least one compound stream:

- stream identification,
- null stream identification,
- sequence generation, and
- R-tag sequence encode/decode.

The requirements of the listener end system are the following out-facing functions on at least one port, and for at least one compound stream:

- stream identification,
- null stream identification,
- sequence recovery,
- at least two instances of the individual recovery function, and
- R-tag sequence encode/decode.

Furthermore, the relay system's requirements include the following functions as in-facing and on at least two ports, for both transmit and receive direction, for at least one stream:

- stream identification,
- null stream identification,
- sequence generation,
- R-tag sequence encode/decode,
- sequence recovery, and
- at least two instances of the individual recovery function.

Moreover, each of the three systems has recommended and optional functions. For example, the splitting function is a recommendation for the talker end system and an option for the relay system. A simple example of a packet's journey through the talker, relay, and listener systems is presented in the following section.

5.2.2 *Packet Flow*

The stream is generated in the talker end system, therefore the flow is described firstly in the talker end system, then relay, and finally in the listener end system. We make here the following assumptions: each relay and listener runs an instance of the individual recovery function, and the listener runs additionally an instance of a sequence recovery function. Moreover, it is assumed that the splitting occurs only in the talker end system, and that merging member streams occurs at the listener end system. However, both functions are

allowed to be invoked in the relay system as well. These assumptions hold in this section only for an easier packet flow description. They do not hold for the entire chapter.

Figure 5.2 represents a packet flow in the network from the perspective of the talker end system. Firstly, a compound stream originates in the talker and is assigned a stream ID (stream_handle). Each packet of the stream is assigned a sequence number, which is encoded into the packet. If the stream shall split, all stream packets are duplicated. The new member streams get new stream_handles and sequence numbers are generated and encoded. Otherwise, the stream is forwarded further to the relay system of the network.

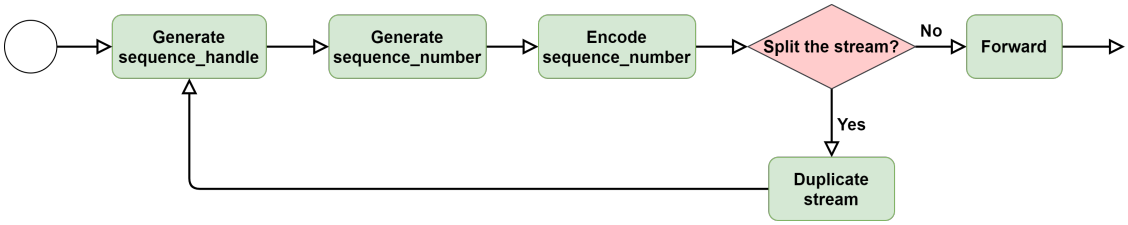

Figure 5.2: Packet processing at the talker end system

The processing of an incoming packet in the relay systems is depicted in Figure 5.3. When the packet is received, it is decoded and, if a duplicate is received previously, it is discarded by the individual recovery function. Otherwise, it is forwarded further to the listener end system.

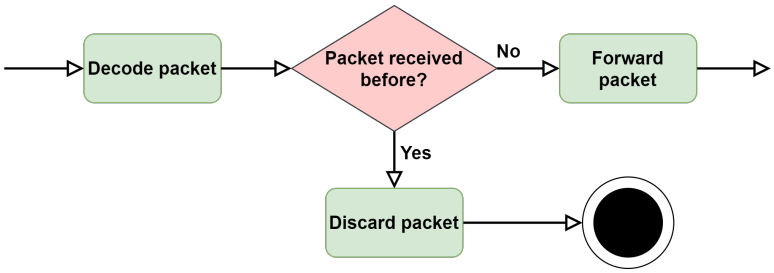

Figure 5.3: Packet processing at the relay system

The packet processing in the listener-end system is represented by Figure 5.4. When the packet is decoded, the same check as in the relay system is performed since the listener end-system also runs an individual recovery function for each member stream. Moreover, at this point in the network, merging the member streams occurs. Therefore, a sequence recovery function eliminates the incoming duplicated packets, and only one copy of each packet remains. This function recognizes the duplicates by checking the entries in the sequence recovery table for which the function is instantiated and examining the stream_handle and sequence_number parameters.

As an additional measure for increasing reliability and easier tracking of possible failures of a network entity, a latent error detection function as a part of the sequencing function can be instantiated. This function examines the numbers of passed and discarded packets, and if the difference between these counters exceeds a defined threshold, then an error state is triggered. With this method, the error informs the system that a failure occurred, and one of the member streams has not reached the listener end system at all.

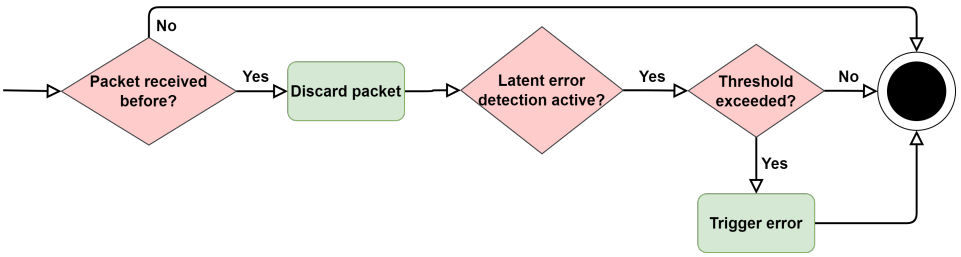

Figure 5.4: Packet processing at the listener end-system

This section briefly explained one possible scenario of a packet flow according to the standard. There are many other possible scenarios with FRER due to the flexibility in positioning its functions.

5.2.3 *Functions of FRER*

In the following, the functions of 802.1CB are presented in terms of their goals and principles of operation. For more detailed description of the functions (events triggering the function, and events being triggered by the function, their variables, as well as the implementations of required algorithms) please refer to the Sections 7.4 - 7.7 of the standard.

STREAM IDENTIFICATION FUNCTION Each system type in the network requires a stream identification function. The basic requirement is having a null stream identification function, which is a passive stream identification. Null stream identification function generates a *stream_handle* sub-parameter for frames passed up the stack based on the frame's destination MAC address and VLAN ID. It does not change any of the packet's other parameters. Other identification types defined by the standard are: source MAC and VLAN stream identification, active destination MAC and VLAN stream identification, IP-based stream identification, and extended stream identification [163].

SEQUENCING FUNCTION This function is required to identify the order of packets in a stream. Furthermore, a network component might resend the same packets due to errors. Therefore, it is necessary to discard the resent packets to avoid network congestion. The sequencing function has two types of component functions: sequence generation and sequence recovery. The sequence generation function is a required function for the talker and relay systems. It generates the sequence_number sub-parameter for the packets coming from higher layers passed down the protocol stack.

The sequence recovery function operates on a merged set of member streams for which the *sequence_number* values are generated by a single instance of a sequence generation function. The function shall be instantiated at the port where the original stream meets its duplicates. For example, with two fully disjoint paths for two member streams, the function shall be instantiated at the listener. The sequence recovery function itself has two component functions:

- The base recovery function discards duplicate packets by checking the *sequence_number* parameter.

- The latent error detection function detects possible deviations by monitoring the objects of a single base recovery function. The assumption taken is that a compound stream having n paths in the network will discard $n-1$ packets for each packet processed by the base recovery function. In case of violation, the latent error detection function signals an error that notifies the system that a member stream has not completely reached the port where the member streams are supposed to be merged, and that a failure in some of the redundant paths has occurred.

The sequence recovery function operates on a merged set of member streams, and therefore, it shall be instantiated at the listener or, in case of partially disjoint paths of member streams belonging to the same compound stream, at the port where the original stream meets its duplicates.

INDIVIDUAL RECOVERY FUNCTION Individual recovery function operates on a single member stream. Its objective is to fulfill the goal of robustness of the standard by removing the repeated packets received from a stuck transmitter. By discovering the erroneous streams early, further pollution of the compound streams is avoided. The individual recovery function operates on a single member stream and the latent error detection function shall not be instantiated. Individual recovery function shall be instantiated at each port of the relays and listener.

SEQUENCE ENCODE/DECODE FUNCTION After the sequence_number is generated by the sequencing function, it is encoded into the packet and later decoded by the sequence encode/decode function. The required encapsulation type is the R-TAG, but two optional formats are additionally defined in the standard. Other encapsulation types are High-availability Seamless Redundancy (HSR) sequence tag, and Parallel Redundancy Protocol (PRP) sequence trailer. The standard requires each of the systems (talker, relay system, and listener) to have the R-TAG sequence encode/decode function. Thus, it needs to be instantiated at all entities of the paths of the streams.

STREAM SPLITTING FUNCTION In order to introduce redundancy to the network, a stream can be duplicated. The role of the stream splitting function is to accept packets from an upper layer and make zero or more copies of the packets, passing them to the next lower layer. The function examines the *stream_handle* sub-parameter of the packet. If this *stream_handle* is configured in the stream splitting table, the stream is duplicated, and at most one duplicate has the same *stream_handle* as the original stream. At the port where the replicated streams meet, the duplicates are filtered by a base recovery function.

5.2.4 *Configuration Tables*

There are two methods by which the functions can be configured in a system:

1. Explicit configuration of entries in the stream identity table, sequence recovery table, and sequence identification table.

2. Autonomic configuration of entries in these tables using the sequence auto-configuration.

The auto-configuration method can be used only with source MAC and VLAN stream identification, and the MatchRecoveryAlgorithm for the recovery functions. However, this

limits the flexibility of the standard. For the purpose of this work, the explicit configuration of the table entries is preferred, and two tables need to be configured additionally: stream splitting table, and sequence generation table. However, we automatically generate these table by ALEVIN as a result of the mapping algorithms. In the following paragraphs, the configuration tables are summarized. For the list of all variables and their detailed explanations, refer to the sections 9.1 and 10.3-10.7 of the standard ([162]).

STREAM IDENTITY TABLE The stream identity table consists of a *tsnStreamIdEntry* object per single stream (specified by the *stream_handle*), and the points in the network where the stream identification method shall be instantiated. The *tsnStreamIdEntry* object is defined by the following: *stream_handle*, input and output port lists that show where the function shall be instantiated, and identification function type. Moreover, depending on the identification function, additional variables are introduced. The main additional parameters for the null stream identification function are the destination MAC address and VLAN ID.

SEQUENCE GENERATION TABLE One entry in the sequence generation table, called as *frerSeqGenEntry*, defines on which streams shall a single sequence generation function operate, regardless from the port on which it will be received. Additionally, the direction of the port where the function is to be instantiated is defined.

SEQUENCE RECOVERY TABLE An entry in the sequence recovery table, called as *frerSeqRcvyEntry*, exists for each sequence recovery or individual recovery function instance in the network. Some of the properties of the *frerSeqRcvyEntry* are: list of *stream_handles* on which the function will operate, list of ports where the function is to be instantiated, if a packet without a *sequence_number* is to be accepted or not, if the function is an individual recovery function, and if latent error detection is active.

SEQUENCE IDENTIFICATION TABLE There is one entry in the sequence identification table, called as *frerSeqEncEntry*, for each port and direction on which the sequence encode/decode shall be invoked. Some of the values to be configured for each entry are: list of streams for which the same encapsulation method is to be used, the port and direction on which the function is to be instantiated, if it is active (used to both encode and decode information of the packet) or passive (only decodes the *sequence_number*), and the encapsulation type.

STREAM SPLIT TABLE An entry in the stream split table, called as *frerSplitEntry*, defines the ports at which a stream splitting function is to be instantiated, and on which set of streams. The *frerSplitEntry* is described by: port and direction on which the function will be invoked, list of input streams which are to be split, and list of output streams to which to split the input streams.

5.3 LINK MAPPING ALGORITHMS

The focus of this chapter is on link mapping algorithms that map a VLi to a path in the SN as a defense mechanism against failures. All algorithms are based on a reliability-aware

k-shortest path algorithm. For each main traffic class in industrial networks, we suggest using a certain defense mechanism. These classes are defined in [164]:

- *Deterministic cyclic traffic class*: calculation of a pair from the k-shortest paths that can satisfy the reliability demand and with the maximal disjointness (branching). This will maximize the reliability that can be achieved by a pair of k-shortest paths, which on the other hand reduces the resource usage.
- *Deterministic acyclic traffic class*: in case no single path can fulfill the reliability demand of the VLi, calculating a pair from the k-shortest paths that can satisfy the reliability demand and with the minimal disjointness (branching).
- *Non-deterministic traffic class*: using only the reliability-aware k-shortest path algorithm. In case no single path from these paths can satisfy the reliability demand of the VLi, it is rejected.

For all classes, the user is assumed to define the appropriate reliability demand value.

5.3.1 *VNE Problem*

The SN is defined as a graph consisting of a set of edges (SLis) $e_s \in E_s$ and a set of vertices (SNos) $n_s \in N_s$. The SNo properties/resources are ID and reliability. The SLi properties/resources are the ID, source node, destination node, available bandwidth, occupied bandwidth, and reliability. The VN is defined as a graph consisting of a set of edges (VLis) $e_v \in E_v$ and set of vertices (VNos) $n_v \in N_v$. The VNo has only an ID demand. The VLi properties/demands are the ID, source node, destination node, bandwidth, traffic class, and reliability. The node mapping in this chapter is only based on one-to-one ID mapping. The link mapping can map a VLi on multiple paths between the source and destination, where each path is verified for the bandwidth demand and the set of paths is verified for the reliability demand.

The reliability of a set of paths is calculated using the Reliability Block Diagram method [165]. We clarify this method for two paths with partial disjointness, as the typical case in this work. In Figure 5.5, we have four blocks. The first block, R1, includes the common links of both paths starting from the source node to the last common node of the two paths in the sequence. R2 and R3 include the distinct links of the paths starting from the spiting point to the merging point. The last block, R4, contains the common links of both paths after merging. The reliability of each block is calculated as a multiplication of reliability resource values of all block entities. Two branches can be recognized: the first branch includes the blocks R1, R2, and R4, whereas the second branch is formed only by block R3.

Path Reliability (PR) for a single path is defined as:

$$PR = \prod_{i=1}^{m} e^i.reliability * \prod_{i=1}^{m} n^i.reliability$$

where m is the length of the path, and n is a destination node of each edge e in the path. For Figure 5.5, the point-to-point reliability (talker -> listener) is calculated as:

$$PR = R1.reliability * (1 - (1 - R2.reliability) * (1 - R3.reliability)) * R4.reliability$$

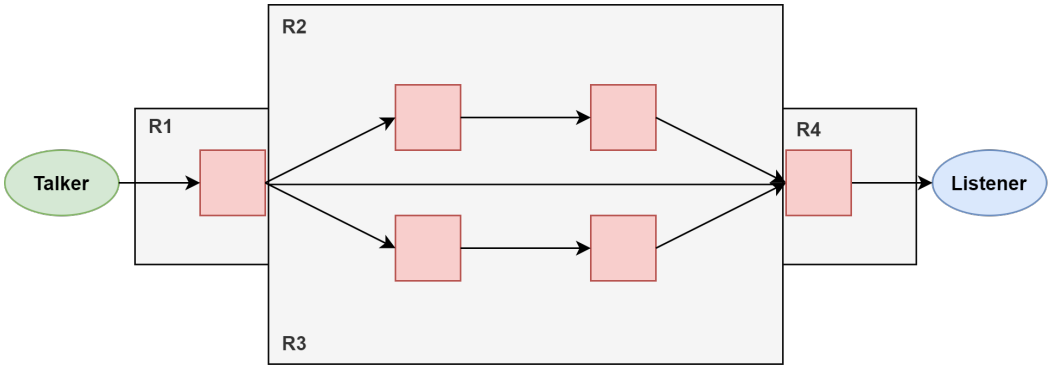

Figure 5.5: Mapping path pair with reliability blocks, adapted from [165]

The RSP algorithm maps the VLis without redundancy. It calculates the paths using the k-shortest paths algorithm with reliability as additional weight (k shortest paths are ordered according to their reliability). The k-shortest paths algorithm is also used for the redundant mapping algorithms, but pairs are selected instead of a single path.

5.3.2 *Maximal Branching*

The MaxBr link mapping algorithm finds a backup path for each $e_v \in E_v$ to be as disjoint from the main mapped path P_s as possible. For each e_v, a set of k-shortest paths is calculated. Each found path is verified for the demands of e_v. When a verified path is found, it is mapped onto the SN. Subsequently, a disjoint backup path is calculated, as shown in Algorithm 8.

The algorithm first removes the already used edges by the main path from the SN except the first and last edges. These are the edges connecting the talker and listener end-points to the network, and finding a disjoint path that excludes them is impossible. The algorithm then searches (and verifies) for k disjoint paths in the reduced substrate SN'. If no path is found or no path is verified, the edges of the main path are iteratively restored to SN'. At each iteration, the algorithm tries again to find and verify one shortest path and stops when this is achieved. Then, all edges from the main path are restored and the disjoint path is returned. As a path pair {main path, disjoint path} is found, the paths are verified for reliability using block digram and then mapped onto the SN if the verification is successful.

5.3.3 *Minimal Branching*

The MinBr link mapping algorithm tries to avoid VLi rejection due to reliability demand when using a single path. It finds from the k shortest paths (ordered according to reliability) the pair with the largest number of common nodes (minimal branching) but still satisfies the reliability demand. With this method, the SN resources are efficiently utilized. If none of the calculated k shortest paths can satisfy the reliability demand, calculating a branch is required as depicted in Algorithm 9.

Algorithm 8: Calculation of maximal branching

```
1  MaxBr ()
2      for e_s ∈ e_v.mainPath.edges do
3          if e_s == e_v.mainPath.edges[0] ∨ e_s == e_v.mainPath.edges[e_v.mainPath.size] then
4              continue
5          end
6          E_s = E_s \ {e_s}
7      end
8      disjointPaths = shortestPaths(e_v.src, e_v.dst, k)
9      for disjointPath ∈ disjointPaths do
10         if pathVerified(disjointPath) then
11             restoreRemovedEdges()
12             return disjointPath
13         end
14     end
15     for e_s ∈ e_v.mainPath.edges do
16         disjointPath = shortestPaths(e_v.src, e_v.dst, 1)
17         if disjointPath == null ∨ ¬pathVerified(disjointPath) then
18             E_s = E_s ∪ {e_s}
19         else
20             restoreRemovedEdges()
21             return disjointPath
22         end
23     end
24 end
```

The minimal branching calculation algorithm iterates over the set of the k paths to find all possible path pairs. Each path pair consists of the main path, which offers the higher reliability of the two paths, and a branch. For each pair, two values are calculated: the reliability using blocks, and the branch length as the number of distinct branch nodes (not shared with the main path). The pairs are sorted with ascending branch length, and the first pair that satisfies the reliability demand is taken.

5.4 NETWORK CONFIGURATION

After executing the link mapping algorithms, we calculate the network configurations in accordance with FRER. We distinguish five node types in the network:

- talker node,
- stream splitting node *SNode*,
- forwarding node,
- streams merging node *MNode*, and
- listener node.

The forwarding nodes, *SNode*, and *MNode* belong to the relay system or TSN network. According to the standard, the required configurations shall be applied to the ports.

Algorithm 9: Calculation of minimal branching

```
1  MinBr ()
2      for path1 ∈ paths do
3          for path2 ∈ paths do
4              if path1!=path2 then
5                  if path1.reliability > path2.reliability then
6                      mainPath = path1
7                      branch = path2
8                  else
9                      mainPath = path2
10                     branch = path1
11                 end
12                 pathPair = (mainPath, branch)
13                 pathPair.reliability = calculateReliability(pathPair)
14                 pathPair.branchLength = calculateBranchLenght(pathPair)
15                 pathPairs.add(pathPair)
16             end
17         end
18     end
19     pathPairs.sort(ascending, branchLength)
20     for pathPair ∈ pathPairs do
21         if pathPair.reliability ≥ e_v.reliability then
22             return pathPairs
23         end
24     end
25     return NULL
26 end
```

However, each function has either an in-facing or out-facing direction at the port. The stream transfer function is defined as a two-port function that transfers service indications on one port to service requests on the other port with all TSN sub-parameters included in the service. An in-facing function is below the stream transfer function on the port and not above the link layer. An out-facing function is below the stream transfer function on the port and above the link layer. When a system has no stream transfer function, all its functions are out-facing. Functions to be invoked in the talker and listener end systems are out-facing functions, and functions to be invoked at the relay system are in-facing functions.

An exemplary scenario is presented in Figure 5.6. Two streams can be observed by *stream_handle* values 10 and 20. In our notation, all input ports of the nodes are assigned the port ID *nodeName*:1. *MNode* has an additional input port, *MNode*:2, at which it receives the stream with *stream_handle* = 20. All output ports are assigned the port ID *nodeName*:2, except for output port *Mnode*:3, and the additional output port *SNode*:3 from which it sends the stream with *stream_handle* = 20. For configuring the network, calculating five configuration tables is required: stream identity, sequence generation, sequence identification, sequence recovery, and stream splitting.

The entry in the stream identity table for the stream with *stream_handle* = 10 is presented in Table 5.1. As shown, the stream identification function is invoked at the out-facing side of the ports only at the talker and the listener end systems. The type of

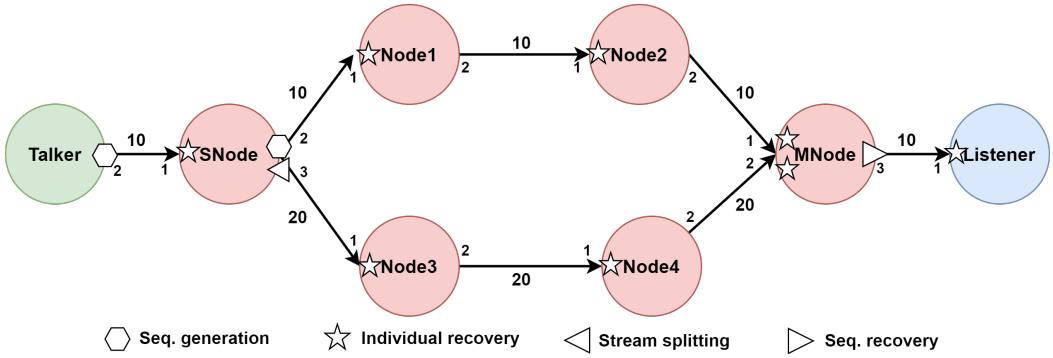

Figure 5.6: Exemplary scenario

the identification function is set as a null stream identification because it is the required identification type common between all systems, according to the standard. Another entry for the same stream can be added with a different identification type. The property *tsnCpeNullDownTagged* can have 3 values: *tagged* (a VLAN tag is required to recognize the frame as belonging to the stream), *priority* (a frame must not have a VLAN tag), and *all* (a frame is recognized as belonging to the stream, independent from the presence of the VLAN tag). The value depends on the desired configuration, and there is no recommendation from the side of the standard. The property *tsnCpeNullDownVlan* stands for the VLAN ID.

Table 5.1: Stream identity table entry for stream 10

Property	Value
tsnStreamIdHandle	10
tsnStreamIdInFacOutputPortList	SNode:2, Node1:2, Node2:2, MNode:3
tsnStreamIdOutFacOutputPortList	Talker:2
tsnStreamIdInFacInputPortList	SNode:1, Node1:1, Node2:1, MNode:1
tsnStreamIdOutFacInputPortLis	Listener:1
tsnStreamIdIdentificationType	Null stream identification
tsnCpeNullDownTagged	All
tsnCpeNullDownVlan	1

The sequence generation table entry specifies an instance of the sequence generation function which is responsible for generating the *sequence_number* of the incoming packets. For our exemplary scenario, the *frerSeqGenEntry* for the first stream is presented in Table 5.2. The direction of the function, specified by a boolean value according to the standard, is out-facing (true) since the stream is generated at the talker node. For the stream with *stream_handle* = 20, the direction would be in-facing (false) since it is generated at the relay system's port.

The sequence recovery table, according to the standard, holds entries for two functions: the sequence recovery and the individual recovery functions. Therefore, for describing the table properly, two entries are explained, one for the individual recovery and another for the sequence recovery function. The entry shown in Table 5.3 describes the individual recovery function for the stream with *stream_handle* = 10 in the relay system. The listener node is omitted from this entry because the function has the opposite direction as the

Table 5.2: Sequence generation table entry for stream 10

Property	Value
frerSeqGenStreamList	10
frerSeqGenDirection	True

relay system, as defined according to the standard, and therefore, a separate table entry is required.

The instance at the listener node has similar entries, except for the port list, and the direction set to *true*. The property $frerSeqRcvyReset$ is a boolean value describing whether the function is to be reset by resetting the *sequence_number* counter. There are two options for the recovery algorithms defined in the standard: vector and match recovery. The vector recovery algorithm accepts the first packet after the recovery reset immediately, and the subsequent packets are accepted under the condition that their sequence_number values are within the range of the previous packet number $\pm frerSeqRcvyHistoryLength$ with the default value of 2, according to the standard.

The match recovery algorithm accepts the first packet after the recovery reset, and the next packet number either matches the last accepted packet number and it is eliminated, or it does not match and it is accepted. The variable *ResetMSec* defines the timeout period in milliseconds after which the recovery function is to be reset. *InvalidSequenceValue* defines a value that cannot be decoded as a *sequence_number*, that is equal to or larger than the defined recovery sequence space, which is a constant value 65 536. The property $frerSeqRcvyTakeNoSequence$ is a boolean indicating whether a packet without a *sequence_number* is to be accepted (true), or discarded (false). The default value in the standard is set to $false$. The next value from the table, $frerSeqRcvyIndividualRecovery$, indicates if the function is an individual or a sequence recovery function, and $frerSeqRcvyLatentErrorDetection$ indicates if a latent error detection function is to be invoked or not.

Table 5.3: Sequence recovery table entry - an individual recovery function instance

Property	Value
frerSeqRcvyStreamList	10
frerSeqRcvyPortList	SNode:1, Node1:1, Node2:1, MNode:1
frerSeqRcvyDirection	False
frerSeqRcvyReset	False
frerSeqRcvyAlgorithm	Match recovery algorithm
frerSeqRcvyHistoryLength	2
frerSeqRcvyResetMSec	3000
frerSeqRcvyInvalidSequenceValue	65536
frerSeqRcvyTakeNoSequence	False
frerSeqRcvyIndividualRecovery	True
frerSeqRcvyLatentErrorDetection	False

The second exemplary entry of the sequence recovery table presented in Table 5.4 describes the sequence recovery function instantiated at port *MNode*:3. Properties with the same values as in the first entry are omitted.

The property *frerSeqRcvyLatentErrorDetection* is set to *true* in order to catch failures in the network. Still, it is up to the network administrator to decide whether to activate it or not. If it is set to true, additional parameters need to be set. The *LatentErrorDifference* is defined as the maximum allowed value of:

$$numDiscardedPackets - (numPassedPackets * (LatentErrorPaths - 1))$$

LatentErrorPeriod, with the default value of 2000 defined in the standard, specifies the time in millisecond between the invocation of two instances of latent error test function, which checks if the number of discarded packets is as expected. *LatentErrorPaths* is the number of paths over which FRER operates for this function instance. *LatentResetPeriod* specifies the time in millisecond between the invocation of two instances of latent error reset function, and default value, according to the standard, is 30 000. The entry configuration for the sequence recovery table is also presented in Algorithm 10.

Table 5.4: Sequence recovery table entry - a sequence recovery function instance

Property	**Value**
frerSeqRcvyStreamList	10, 20
frerSeqRcvyPortList	MNode:3
frerSeqRcvyIndividualRecovery	False
frerSeqRcvyLatentErrorDetection	True
frerSeqRcvyLatentErrorDifference	10
frerSeqRcvyLatentErrorPeriod	2000
frerSeqRcvyLatentErrorPaths	2
frerSeqRcvyLatentResetPeriod	30000

Sequence encode/decode function inserts the *sequence_number*, generated by the sequence generation function, into the packet, and extracts it from the packet. The function is required for each entity included in the path of the packet, except with a stream identification method that already inserts and extracts the *sequence_number*, such as the active destination MAC and VLAN stream identification. For each instance of the function, an entry in the sequence identification table is created. Two entries for our exemplary scenario are presented in Table 5.5.

The first entry describes the properties of the function at port *Talker:2*, and the second entry describes the function instance at port *SNode:1*. The list of streams for which an instance of the function is to be instantiated at the port noted in *frerSeqEncPort*, in the given direction and encapsulation type, is specified by *frerSeqEncStreamList*. The direction is specified in *frerSeqEncDirection* with a boolean value, indicating whether the function is on the out-facing side of the port. For our scenario, for the port *Talker:2*, it shall be set to True, and for the *SNode:1* to False since it is a part of the relay system. The property *frerSeqEncActive* indicates if the function is to be used to encode and decode packets (and it is set to True, as it is the case at port *Talker:2*) or only for decoding (as it is the case at port *SNode:1*).

Algorithm 10: Configuration of a sequence recovery table entry

```
1  ConfigureSeqRecovery ()
2      frerSeqRcvyEntry.frerSeqRcvyStreamList = streamList
3      for stream ∈ streamList do
4          for port ∈ stream.ports do
               // set the entry for the listener port
5              if port == listener.port then
6                  frerSeqRcvyEntry.frerSeqRcvyDirection = true
7              else
                   // set the entry for the relay port
8                  frerSeqRcvyEntry.frerSeqRcvyDirection = false
9              end
10             frerSeqRcvyEntry.frerSeqRcvyPortList.add(port)
11         end
12         frerSeqRcvyEntry.frerSeqRcvyReset = false
13         frerSeqRcvyEntry.frerSeqRcvyAlgorithm = MatchRecovery
14         frerSeqRcvyEntry.frerSeqRcvyResetMSec = 3000
15         frerSeqRcvyEntry.frerSeqRcvyInvalidSequenceValue = 65536
16         frerSeqRcvyEntry.frerSeqRcvyTakeNoSequence = false
17         if individualRecovery == true then
18             frerSeqRcvyEntry.frerSeqRcvyIndividualRecovery = true
19             frerSeqRcvyEntry.frerSeqRcvyLatentErrorDetection = false
20         else
21             frerSeqRcvyEntry.frerSeqRcvyIndividualRecovery = false
22             if latentErrorDetection == true then
23                 frerSeqRcvyEntry.frerSeqRcvyLatentErrorDetection = true
24                 frerSeqRcvyEntry.frerSeqRcvyLatentErrorDifference = 10
25                 frerSeqRcvyEntry.frerSeqRcvyLatentErrorPeriod = 2000
26                 frerSeqRcvyEntry.frerSeqRcvyLatentErrorPaths = 2
27                 frerSeqRcvyEntry.frerSeqRcvyLatentErrorResetPeriod = 30000
28             end
29         end
30     end
31 end
```

Note that at *SNode* three instances of the function are required to be invoked, one at each port, two active encode/decode functions at the output ports, and one passive at the input port. Next, the encapsulation type is set by $frerSeqEncEncapsType$. The standard defines three possible encapsulation types: R-TAG ($value = 1$), HSR sequence tag ($value = 2$), and PRP sequence trailer ($value = 3$). The R-TAG is required by the standard for the three system types, therefore, it shall be used as the default value. Other types can be used only as an addition to the R-TAG. In case of a passive function ($active = false$) with any of the two optional encapsulation types, the $frerSeqEncPathIdLanId$ shall be set as the path ID or LAN ID. For more details about the types, refer to sections 7.8, 7.9, and 7.10 of the standard.

The stream split function is configured in the stream split table. The entry values for our exemplary scenario are given in Table 5.6. The port at which the function is to be instantiated is specified by $frerSplitPort$, which is in this case *SNode*:1. The $frerSplitDirection$ indicates whether the function is at the out-facing side of the port.

Table 5.5: Sequence identification table - an active and a passive function instance

Property	Value 1	Value 2
frerSeqEncStreamList	10	10
frerSeqEncPort	Talker:2	SNode:1
frerSeqEncDirection	True	False
frerSeqEncActive	True	False
frerSeqEncEncapsType	1	1
frerSeqEncPathIdLanId	0	0

Since, in our case, the function is in the relay system, the direction is defined as $false$. The stream to be split, noted in $frerSplitInputIdList$, has a $stream\_handle = 10$, and is split into streams with $stream\_handles$ 10, and 20, as noted in the $frerSplitOutputIdList$.

Table 5.6: Stream split table entry

Property	Value
frerSplitPort	SNode:2
frerSplitDirection	False
frerSplitInputIdList	10
frerSplitOutputIdList	10, 20

5.5 IMPLEMENTATION

As mentioned before, we have two main goals: to improve the resilience of VNs with efficient methods, and to integrate the embedding results with the Std. IEEE 802.11CB FRER. Firstly, the required tools for implementation are presented, then further implementation details of the main tasks are provided. For implementing the reliability methods and validating the configurations of the network via a simulator, two frameworks have been used: ALEVIN and TSimNet. In the following sections, the most important features of the frameworks are described.

5.5.1 *Link Mapping Algorithms*

Most of the implementation of the reliability methods is performed in the framework ALEVIN, including: the implementation of the link mapping algorithms, their evaluation, and computing and exporting the network configurations. Firstly, ALEVIN parses JSON definition files for the generation of the SN and VNs. The format of the definition file for the SN is presented in Listing 5.1. The SN consists of two types of elements: nodes and links. A node has the following properties: ID, x and y coordinates, and resource attribute which is the node's reliability in this case. A link of the SN is defined according to Listing 5.2. The *linkResource* property holds the following information: the maximum, available, and reserved bandwidth, and the link's reliability. The link properties additionally include the IDs of the source and destination SNos and ports.

Listing 5.1: SN JSON definition format

```
"network":{
    "elements":{
        "nodes":[
            {
                "id": "host1",
                "X": 0.323,
                "Y": 3.172,
                "resource": {"reliability": 0.99 }
            },
            ...
        ],
        "links": [{ .... }]
    }
}
```

Listing 5.2: SLi JSON definition format

```
"links":[
    {
        "id": "host1:1/host2:1",
        "linkResource": {
                "maximumBandwidth_BitsPerSec": "1000000000",
                "availableBandwidth_BitsPerSec": "1000000000",
                "reservedBandwidth_BitsPerSec": "0",
                "linkReliability": 0.99
                }
        "src": {
            "srcNodeId": "host1",
            "srcPort": "host1:1"
        },
        "dst": {
            "dstNodeId": "host2",
            "dstPort": "host2:1"
        }
    },
    ...
]
```

The VN JSON definition format is presented in Listing 5.3. Each VN in this reliability work represents a network slice deployed in a factory network to represent resource reservations for the VLis [6]‡. The *sliceEndPoints* property holds the {Talker, Listeners} tuple, and the property *reliabilityAlgorithm* refers to the link mapping algorithm to be used. We define a slice per talker and our VNE tool creates a VLi from this talker to each listener.

Listing 5.3: VN JSON definition format

```
[
    {
        "sliceId": "Video1",
        "sliceEndPoints": [
```

```
            {"Talker": "Camera161",
            "Listeners": ["Video_Server121","Video_Server221"]},
            ...
        ],
        "reliability": "0.8",
        "reliabilityAlgorithm": "MinBr",
        "demandedBandwidth":"10000000"
    },
    ...
]
```

After the network definition files are read and the network stack, consisting of the SN and slices, is created, the slices are mapped using the algorithms defined in Section 5.3. Finally, the configuration tables are computed and exported in XML format for the network simulation framework TSimNet.

5.5.2 *Network Configuration*

TSimNet is an open-source network simulation framework based on OMNET++ and INET frameworks. It implements the following parts of TSN standards family: 802.11Qbu Frame Preemption, 802.11CB FRER, and 802.11Qci Per-Stream Filtering and Policing [166].

A project within this framework requires a .ned file and an .ini file to be defined. The .ned file is the network definition file: the nodes and links are defined, and names of ports are automatically assigned. Additionally, the .ned file includes a graphical representation of the network, and for that purpose, coordinates of the nodes are required to be set as well. The network can be either manually written using the NED language or imported from one of the other file formats such as CSV, JSON, and XML; each of the file formats is precisely defined. For the purposes of this work, the XML file format is used because it is the default file format of ALEVIN and, therefore, the required changes of the exported network stack from ALEVIN for the simulator are minimal.

Further, the .ini file is used for the general network configurations such as the address resolution protocol, if UDP or TCP is used, packet sending interval, and length of the message. The file is also used to define the TSN capabilities of the network such as recovery functions, encapsulation types, stream splitting and merging, and others. Besides the .ini file, the main class responsible for the network configuration is the $TSNConfigurator$. This class is responsible for network configurations by reading the topology, computing the interface and routing tables, configuring addresses of the network entities, and computing static routes or importing TSN routes from a file.

In order to configure the network according to the calculated paths, the simulator requires $tsn-route$ to be computed and imported. In case of no $tsn-routes$, the $TSNConfigurator$ calculates static routes as unweighted shortest paths. Therefore, computation of $tsn-route$ elements is equally important as the configuration tables from the standard in order to run the simulation properly. This element is required for the simulator, but it is not specified by the standard. The element is represented as:

```
<tsn-route src="host1" dst="host2" path="node1 node2" stream-id="1"
sequence-filtering="enabled" rate="200000" burst="40000"/>
```

As shown, the $tsn - route$ tag has a source, destination, path, and stream-id (which is the *stream_handle*). The other attributes are not relevant for this work. For one mapped VLi with redundancy, three $tsn - route$ elements are required: from the talker to the node where splitting occurs, and two paths from splitting node to the listener. Besides the $tsn - route$ tag, other required configuration entries are implemented as described in Section 5.4.

After calculating the configuration tables by ALEVIN, they are exported to be imported into the network simulator. The simulator imports the configuration as an XML file, as a property of the $TSNConfigurator$ class. The file initially includes the network mask, addresses, and the $tsn - routes$. For the purpose of this work, the configurator is extended to read the computed configuration tables. The FRER capabilities of the simulator are implemented in a manner that the *nodes* are configured differently from the standard, which configures the specific *ports*. Therefore, properties such as the direction of a function (in- or outing-facing side) in a port cannot be set. Moreover, not all functions can be configured, such as sequence encode/decode function, since the simulator implements them in the structure of the port itself. We modified the simulator to configure the stream splitting function and the recovery functions (the individual and sequence recovery).

The general method of accessing entries of the configuration tables is presented in Algorithm 11. The configuration file is read and, for example, for the stream split table each split entry is found in the file, and its attributes are assigned to certain variables. Further, the matching node from the topology holding the specified port of the entry is found and the configuration is applied.

Algorithm 11: Setting the configuration tables

```
1  SetConfTables ()
2      XMLElementList entryElements = config.childrenByTagName("entry")
3      for entryElement ∈ entryElements do
4          readEntryAttributes()
5          host = entry.port
6          for i = 0; i<topology.numNodes; i++ do
7              Node node = topology.nodes[i]
8              if node ≠ host then
9                  continue
10             end
11             setConfigurationTable()
12         end
13     end
14 end
```

Besides setting the variables from the configuration tables in the nodes, additional properties are required to be set due to the structure of the simulator. For the stream split table, a property of type $TSNSplitTable\_t$ is required. This table holds the pairs of $\{input\ stream,\ output\ stream\_handle\ list\}$. For example, in the case of a stream with stream_handle 1 is to be split into streams 2 and 3, the entry would be {1, {2, 3}}. Moreover, a forwarding entry for the $TSNForwardingTable\_t$ is set. Each entry is defined by the stream to be forwarded and the forwarding port of the node on which the configuration is applied. In case of a stream 2 to be forwarded on port 24, the entry would be {2, 24}.

For the sequence recovery table, a *TSNMergeTable_t* is required to be set. It consists of pairs of {*the stream to be merged, the stream into which to merge*}. For example, if stream 2 is required to be merged into stream 1, the entry in the merge table is {2, 1}. It is to note that the simulator does not implement the latent error detection function, which is part of the sequence recovery function.

5.6 EVALUATION OF LINK MAPPING ALGORITHMS

5.6.1 *Metrics*

The performance of the VNE algorithms is assessed in terms of acceptance ratio, SN utilization, and path reliability. *Acceptance Ratio (ARa)* shows the ratio of mapped slices to the total number of slices:

$$ARa = \frac{Number\ of\ mapped\ slices}{Number\ of\ requested\ slices}$$

We define a new application *Resource Utilization (RU)* metric that represents the degree to which a certain link mapping algorithm improves (decreases) the total length of paths (number of links) used to embed the accepted application slices. It is defined as:

$$RU = ARa * \frac{Total\ demanded\ bandwidth}{Total\ reserved\ bandwidth}$$

Where ARa is the acceptance ratio of the application, the demanded bandwidth is the total demanded bandwidth of all VLis of all slices of the application, and the reserved bandwidth refers to the total reserved bandwidth for all application slices. For example, if we have six video slices and two VLis in each slice and with a bandwidth demand of 1, we have a total bandwidth demand of 12. We assume in this metric that the bandwidth demands of the same application are very close. If RSP algorithm accepts 2 out of 6 slices ($ARa = 0.33$), and uses 24 SLis for mapping the 4 accepted VLis (average path length is 6 hobs, reserved bandwidth is 24), then $RU = 0.33 * (12/24) = 0.16$. If MaxBr algorithm uses doubled average number of SLis per VLi (12) and accepts all slices, then $RU = 1 * (12/144) = 0.08$. In this case, RSP doubles the resource utilization. Although this metric is affected by the distances among application SNo, it is useful in comparing different link mapping algorithms with the same application. The path reliability, as a property of a mapped path, is as defined in Section 5.3.1. Additionally, the runtime of the algorithms is compared.

5.6.2 *Evaluation SN Topology*

As shown in Figure 5.7, the factory topology includes four sub-networks: the plant backbone with multiple connected rings, and three cells with different topologies. The plant backbone includes two Internet nodes, three video servers, two cameras, three PLCs, and Human-Machine Interface (HMI) devices. The cells include bridges to the plant backbone, PLCs, cameras, and general IO devices. The reliability value of all SN entities is 0.99. The IDs of backbone bridges are numbered with the ring number first then the bridge order in the ring. The IDs of cells bridges are numbered with the cell number first then the order of the bridge in the line/ring. All devices are numbered with

the numeric ID of the attached bridge then the order of this device for this bridge. For this evaluation, the parameter k of the k-shortest path algorithm is set to 3 to exclude very long paths and, at the same time, allow multiple path pairs.

We define slices from five application types: video, Internet, control, PLC, and best effort (BE). Each slice connects a talker to a set of listeners. The purpose of having multiple listeners in a slice is building a tree for the slice in another work that is not included in this thesis. A video slice transfers video from a camera to multiple video servers. An Internet slice connects an Internet gateway to multiple HMIs. A BE slice connects a HMI to multiple IO devices for configuration and monitoring. A PLC slice connects a backbone PLC to multiple cell PLCs. A control slice connects a cell PLC to multiple IO devices from the same cell. The assumed reliability demand of each application type and the {Talker, Listeners} tuples are presented in Table 5.7.

Table 5.7: Application types and their descriptions

Type	Talker	Listeners	Reliability demand
Video	Camera	Video servers	0.8
Internet	Internet	HMIs	0.7
BE	HMI	IO devices	0.7
Control	Cell PLC	IO devices from the same cell	0.85
PLC	Plant PLC	Cell PLCs	0.85

The slice definition file contains in total 30 slices with 6 slices from each application (video, Internet, BE, control, PLC). The evaluation scenario has been run 30 times for each of the three link mapping algorithms, and in each run, a new order of VNs/slices arrival is used. The reason behind different orders of slice arrivals is that some slices need more resources (for example, the video slices need more bandwidth, and control slices require more reliable paths). Thus, slices might be mapped on different paths in each run. However, the confidence intervals of 95% in such a fixed topology were too small to be seen in the results figures.

5.6.3 *Results and Discussion*

In this section, we present and discuss the evaluation results that compare the three algorithms in terms of acceptance ratio and resource utilization per application, path reliability per VLi, and runtime per slice.

5.6.3.1 *Acceptance Ratio*

The effect of redundant paths is visible from the values of acceptance ratio in Figure 5.8. RSP maps 73% of requested slices, whereas the MinBr and MaxBr algorithms both achieve 100% mapping. The reason is low admission ratio with RSP for the PLC and video slices (2 from 6 slices mapped (33%)) that have high reliability demand (0.85 and 0.8) and relatively some long paths from the cell cameras to the video serves and from cell PLCs to the backbone PLCs. The path length decreases the reliability, as discussed in

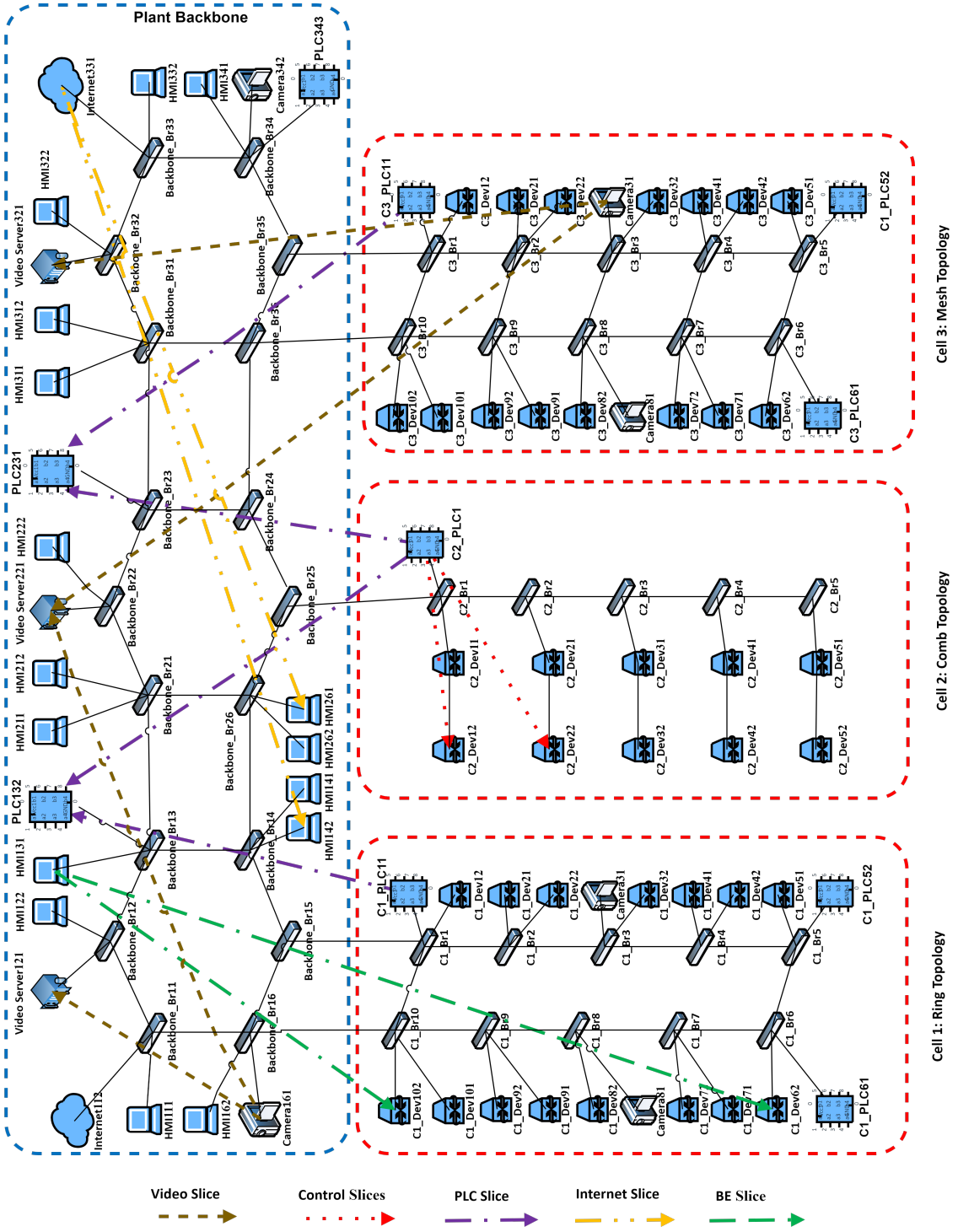

Figure 5.7: Evaluation scenario - SN topology

Section 5.3.1. In this case, the branching can increase the reliability but still cannot always guarantee 100% mapping.

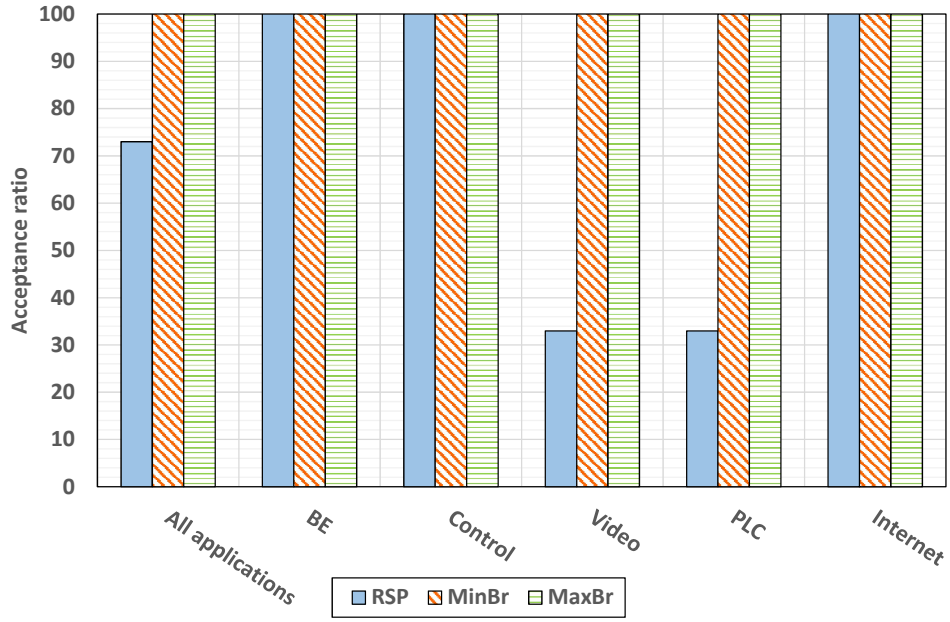

Figure 5.8: Acceptance ratio per application type

5.6.3.2 *Path Reliability*

We observe the path reliability for each VLi of the evaluation scenario. The average values for all VLis from each application are presented in the following figures. The path reliability value is calculated from both VLis and VNos and is 0 for rejected VLis. Since we have rejected PLC and video slices with RSP, we see 0 reliability for some PLC and video VLis when RSP is used. In general, the path reliability is improved by MinBr and the best with MaxBr. However, we see some overlapping points (similar reliability value) in the following cases:

- For RSP and MinBr, when RSP is able to map the VLi on a single path that satisfies the reliability demand. In this case, MinBr will not try to branch and will also use the same single path.
- For MaxBr and MinBr, when the path pair with minimal branching that can satisfy the reliability demand is the same pair with maximal branching.
- For the three algorithms, when there is only one possible path for the VLi, and this path satisfies the reliability demand. This might be due to the topology itself or available bandwidth. In this case, MinBr will not try to branch, and MaxBr will not find a backup path.

Figure 5.9 shows the path reliability of the VLis belonging to the video slices. We see for RSP that reliability = 0 for ~66,67% of the VLis, implying that these are not mapped.

For these rejected VLis, both branching algorithms can find a mapping that satisfies the reliability demand. However, MaxBr improves the reliability with the maximum difference of ~0.094 to MinBr.

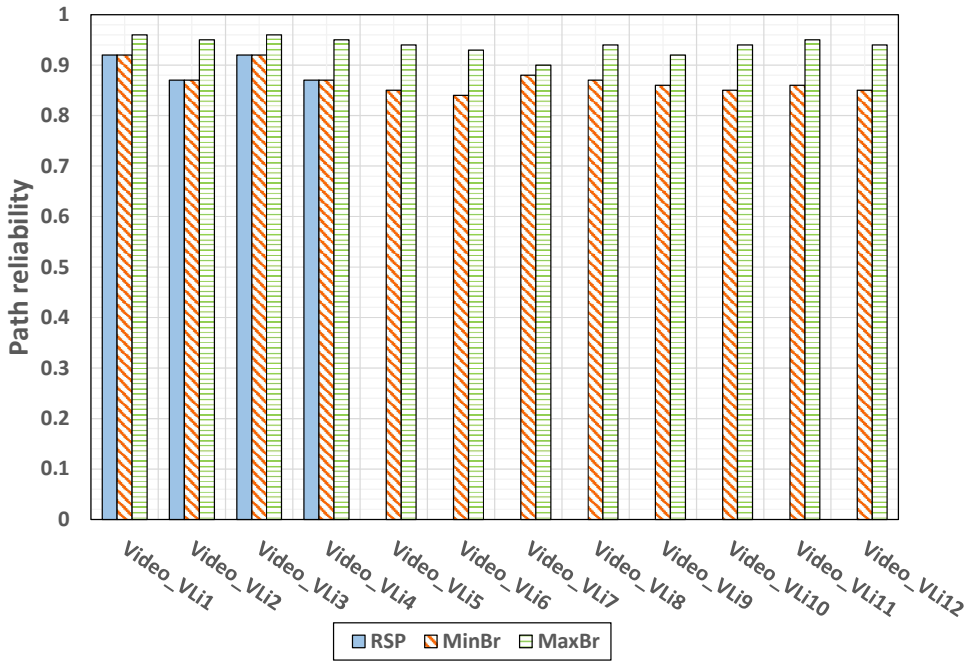

Figure 5.9: Path reliability per VLi for video slices

Figure 5.10 shows the path reliability of the VLis belonging to the Internet slices. Since the reliability demand is relatively low and all end-nodes are in the backbone, there is always a possibility to find a single mapping path and we see a complete overlapping between RSP and MinBr. However, MaxBr highly improves the reliability (maximum difference to MinBr is 0.16). This improvement is due to the higher possibility of path (partial) disjointness in the backbone.

In Figure 5.11, we see that the RSP algorithm maps all BE VLis due to relatively low reliability demand; therefore it overlaps with MinBr completely. Moreover, we also notice an improvement when using MaxBr with the maximum increase in reliability of ~0.11. However, this improvement is less than what we see for the Internet slices since the possibility of path (partial) disjointness is less between the cells and backbone than inside the backbone.

For the control slices, from Figure 5.12, we can observe the overlapping of the three algorithms for two VLis from Cell 2 since there are only single paths among IO devices and the PLC. Although the reliability demand is relatively high for control slices, the paths inside the cells are relatively short, and RSP (and MinBr) is always able to find a single mapping path. In cell 1, MaxBr maps the other direction in the ring as a branch. In most cases of Cell3, the length of the branch is equal to the length of the main path. MaxBr improves the reliability compared to RSP with a maximum of 0.09.

Figure 5.13 shows the path reliability for the PLC VLis. We note that RSP cannot map 66% of the slices due to some long paths (between backbone PLCs and lower cell PLCs) and high reliability demand. For the VLis that RSP can map (for upper cell PLCs), the

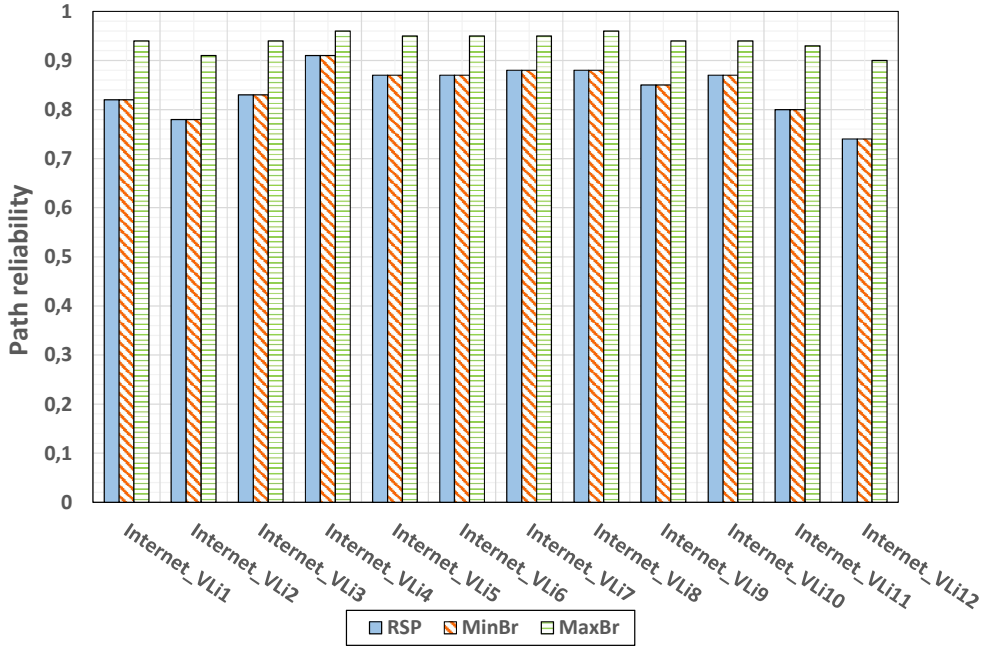

Figure 5.10: Path reliability per VLi for Internet slices

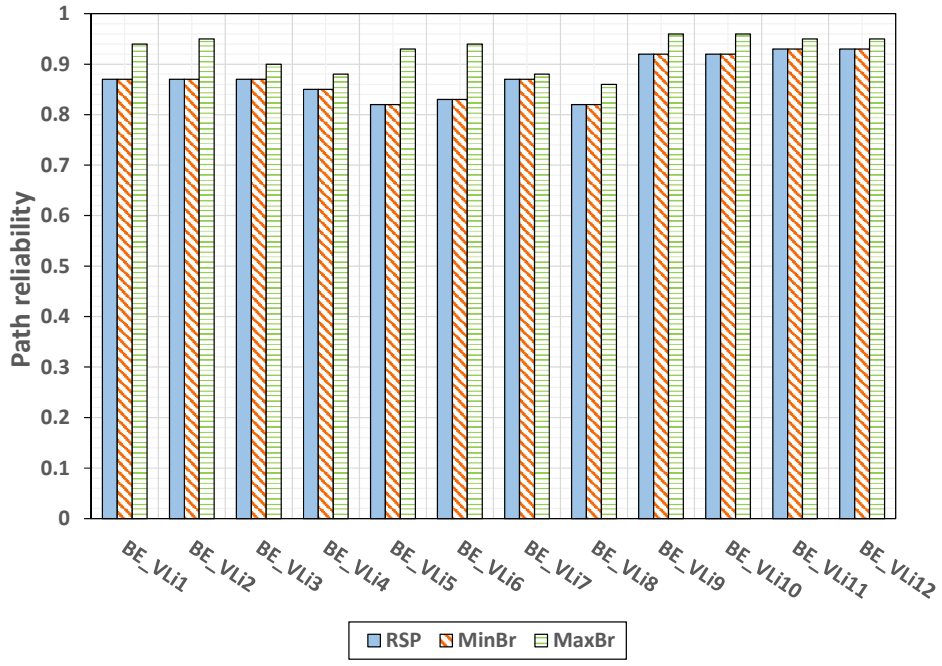

Figure 5.11: Path reliability per VLi for BE slices

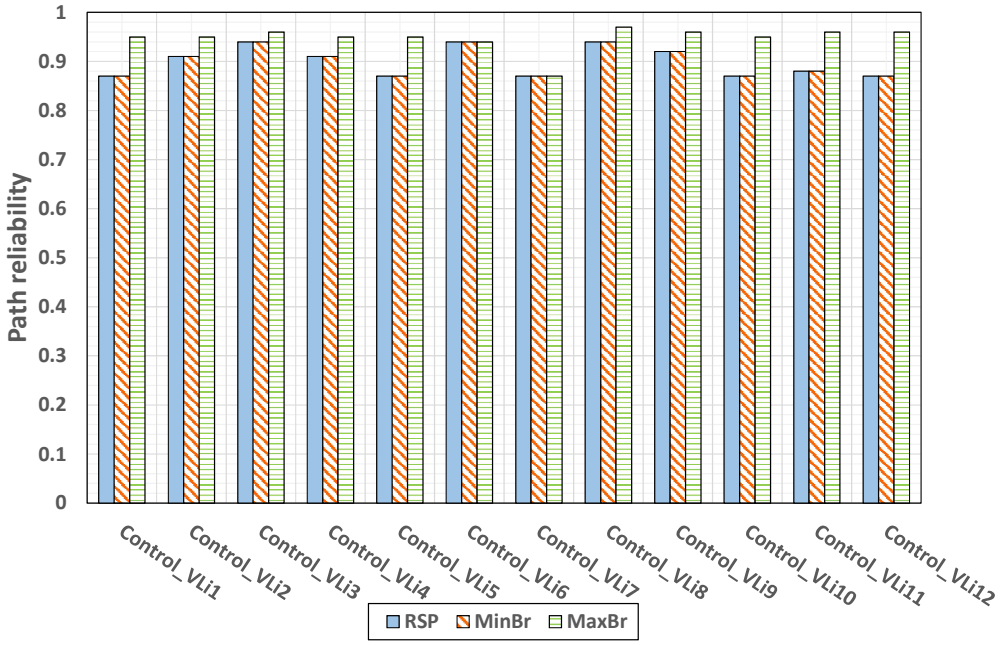

Figure 5.12: Path reliability per VLi for control slices

results of RSP and MinBr overlap, and the minimal and maximal branches have close length. For the other VLis (for lower cell PLCs in cells 1 and 3), MaxBr can use highly disjoint paths and improve the reliability with a maximum of 0.067, which is smaller than other applications due to the length of paths.

5.6.3.3 *Resource Utilization*

The results of the RU metric presented in Figure 5.14 give an overview of the efficient usage of resources by the different link mapping algorithms for each application. For all applications, we consider the bandwidth demand as 1 unit and each reservation on an SLi as 1 unit. This is true even if the real values are different according to the resource utilization equation. Each application has 6 slices with two VLis leading to a bandwidth demand of 12. The total reserved bandwidth in this case is the number of mapped VLis multiplied with the average application path length when using a certain algorithm.

For video slices and RSP, 4 VLis are mapped with average path length of 6 leading to $RU = 0.33 * (12/(4 * 6)) = 0.165$. MinBr maps all VLis with average path length of 6 for the 4 VLis that RSP can map, and of 9 for the remaining 8 VLis due to a branch of additional 3 VLis. Then the total average path length for video slices with MinBr is $(4 * 6 + 8 * 9)/12 = 8$, leading to $RU = 1 * (12/(12 * 8)) = 0.125$. MaxBr maps all VLis with average path length of 10 due to a branch of additional 4 VLis in average leading to $RU = 1 * (12/(12 * 10)) = 0.1$. This average branch length is due to the topology that forces some common hobs among the k-shortest paths.

For Internet slices, the reliability demand is relatively low and RSP (and MinBr) are able to map all VLis with an average path length of 8.5 (all in the backbone but we connect the HMIs to the distant Internet gateway to compare the algorithms). The resource utilization

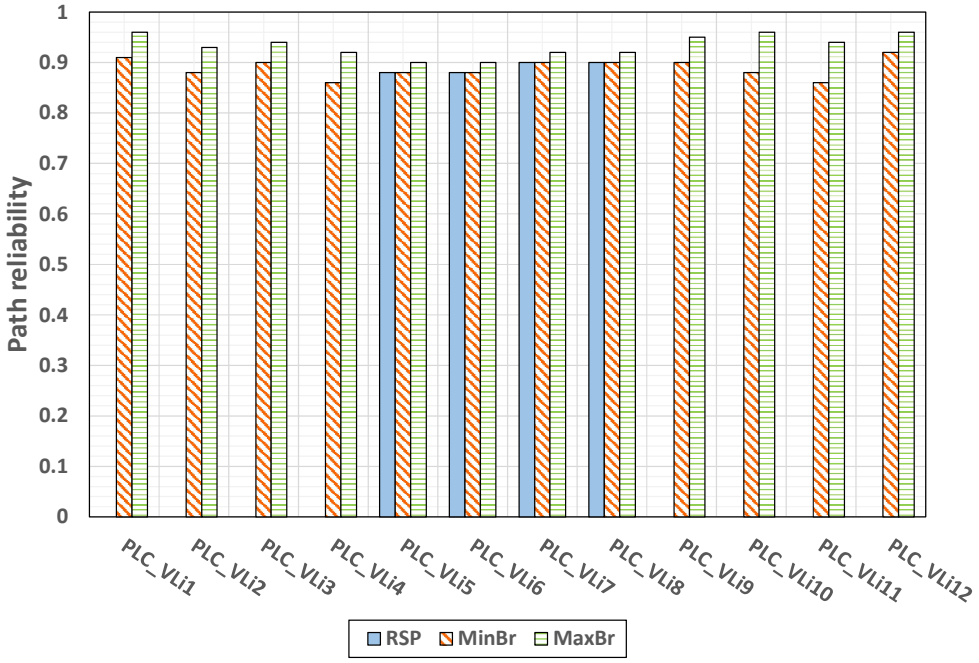

Figure 5.13: Path reliability per VLi for PLC slices

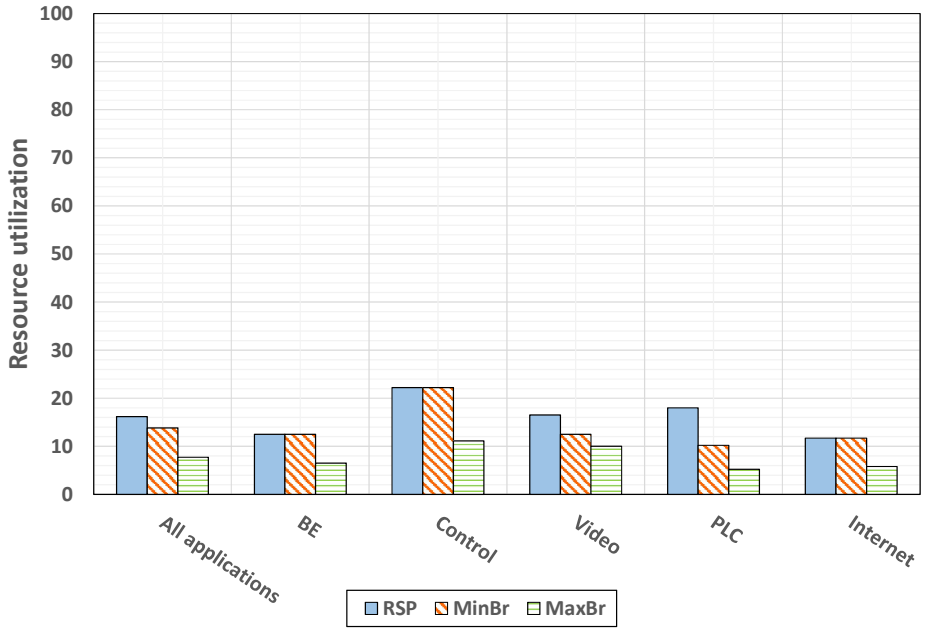

Figure 5.14: Resource utilization per application type

for RSP and MinBr is $RU = 1 * (12/(12 * 8.5)) = 0.117$. MaxBr in such a backbone and long path adds branches of similar length. The two SLis to the end-nodes are replaced with the two SLis that split and merge the parallel branch. For this reason, MaxBr maps all VLis with an average path length of 17, and $RU = 1 * (12/(12 * 17)) = 0.058$

For BE slices, the reliability demand is relatively low and RSP (and MinBr) are able to map all VLis with an average path length of 8 (different path length values between the cells and backbone HMIs). The resource utilization for RSP and MinBr is $RU = 1 * (12/(12 * 8)) = 0.125$. For the lower IO devices of cells 1 and 3 to the backbone HMIs, there a possibility to map long maximal branches. From the upper IO devices of cells 1 and 3 and all cell 2, and with k=3, the maximal branching mainly happens inside the backbone. In this case, MaxBr will not select the longest branch in the backbone but the shortest. The logic of MaxBr removes the SLis of the main path and then applies k-shortest path again on the SN. When no path is found, the SLis of the main path are restored sequentiality until a branch is found. This means that if multiple branches have the same number of common SLis with the main path, MaxBr will choose the shortest branch, and maximal here means shortest path with the maximal disjointness from the main path or least number of common SLis. In average, the branch length with MaxBr and BE slices is 7.3 and $RU = 1 * (12/(12 * (7.3 + 8)) = 0.065$

For control slices, we deploy slices with short paths between the IO devices and PLCs (average path length is 4.5). For RSP and MinBr, $RU = 1 * (12/(12 * 4.5)) = 0.222$. MaxBr uses the same single path in cell 2 for two VLis and the ring size plus the edge nodes connections in cell 1 (12). In cell 3, MaxBr adds as a branch one bridge (2 SLis) for paths with length of 4, and two bridges (3 SLis) for the paths with length of 5. We have 3 slices in cell 1, 1 slice in cell 2, and 2 slices in cell 3. Then the average path length of control slices with MaxBr is $(12 * 6 + 4.5 * 2 + 7 * 4)/12 = 9$, and $RU = 1 * (12/(12 * (9)) = 0.11$.

For PLC slices, we have two slices that RSP (and MinBr) can map and which connect the upper cell PLCs with an average path length of 5.5. The utilization for RSP is $RU = 0.33 * (12/(4 * 5.5) = 0.18$. The other four slices connect the lower PLCs with an average length of the direct path of 9.5. MinBr adds 2.5 SLis from the backbone in average as branches and the average path length (4*5.5+8*(9.5+2.5))/12 =9.8 and $RU = 1 * (12/(12 * (9.8)) = 0.102$. MaxBr adds mostly equal length branches (9.5), and $RU = 1 * (12/(12 * 19) = 0.053$.

To average resource utilization for all applications and for RSP, MinBr, and MaxBr is 0.162, 0.138, 0.077, respectively. In general, MinBr improves the resource utilization with 6% compared to MaxBr and achieves a similar acceptance ratio and average maximum reduction of reliability per application of 0.16. However, the specific values depend on the application and SN topology and nature of paths. The differences between RSP and MinBr depend on the reliability demands and distances among end-nodes, which affect the possibility of using one path.

5.6.3.4 *Runtime*

Figure 5.15 shows the runtime metric results measured by ALEVIN for two representative slices from each application. The runtime of RSP and MinBr is very close except when there is rejection by RSP (in slices video2 and PLC2). In this case, RSP takes a bit less time than the case of accepting the slice since the k-shortest paths are also found and checked for reliability but no path is really mapped. In the same case, MinBr takes a bit

longer time than its time in the case of RSP admission, but clearly longer time than RSP here since it finds a pair of paths with the minimal branching. The main time-consuming operations are Dijkstra, and in MinBr finding the number of common nodes in each path pair, and in MaxBr removing the main path from the topology and restoring it sequentially until the k-shortest path algorithm finds a path. However, calculating the reliability of a single path or pair of paths and sorting paths/pairs according to reliability or number of common nodes are less time-consuming operations.

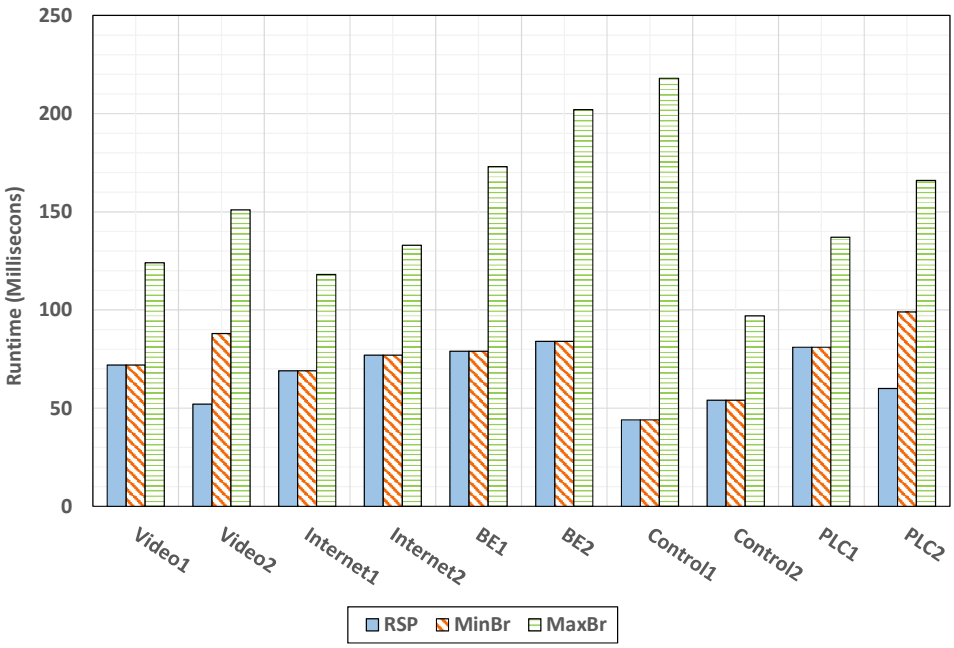

Figure 5.15: Runtime per slice

In general, the exact runtime of RSP is affected by the distance among end nodes. For this reason, we have close runtime for video and Internet slices that have close direct path lengths in our evaluation setup. Furthermore, we have a bit longer paths for BE and PLC slices, and the shortest paths for control slices. These differences can be seen in the RSP runtime for these applications.

The runtime of MaxBr depends on the length of paths and possibility of disjointness. However, we notice for control slices that there is either a small difference in cells 1 and 3 where finding the maximal branch is fast, or a large difference when there is only a single path (in cell 2) and MaxBr restores all edges trying to find a path, but at the end it uses the same path.

5.6.3.5 *Discussion*

Based on our findings, we suggest a mapping among the three algorithms and the mentioned traffic classes. RSP shall be used for non-deterministic traffic that usually demand lower reliability. MinBr shall be used for deterministic aperiodic traffic with unknown arrival time and high reliability demand with low bandwidth reservation. MinBr can satisfy the reliability while saving resources. MaxBr shall be used for deterministic periodic

traffic that has the highest reliability and bandwidth demand. MaxBr can increase the reliability with accepted costs since is finds the maximal branching in a set of k-shortest paths.

5.7 TESTING NETWORK CONFIGURATION

In this section, the calculated configuration tables according to FRER are validated. The validation is performed in the aforementioned network simulator, TSimNet. The SN topology used for the purpose of evaluation is the same topology from Section 5.6.2. For calculating the configuration tables, we use the slice definition format shown in Listing 5.4.

Listing 5.4: Slice JSON definition format

```
[{
    "sliceId": "Video",
    "sliceEndPoints":
        [{"Talker": "C3_Camera31","Listeners": ["Video_Server121"]}],
    "reliability": "0.8",
    "reliabilityAlgorithm": "MinBr"
}]
```

The slice has only one {Talker, Listeners} tuple and only one listener for easier graphical representation in the simulator. In Figure 5.16, only the mapped paths of the substrate are shown.

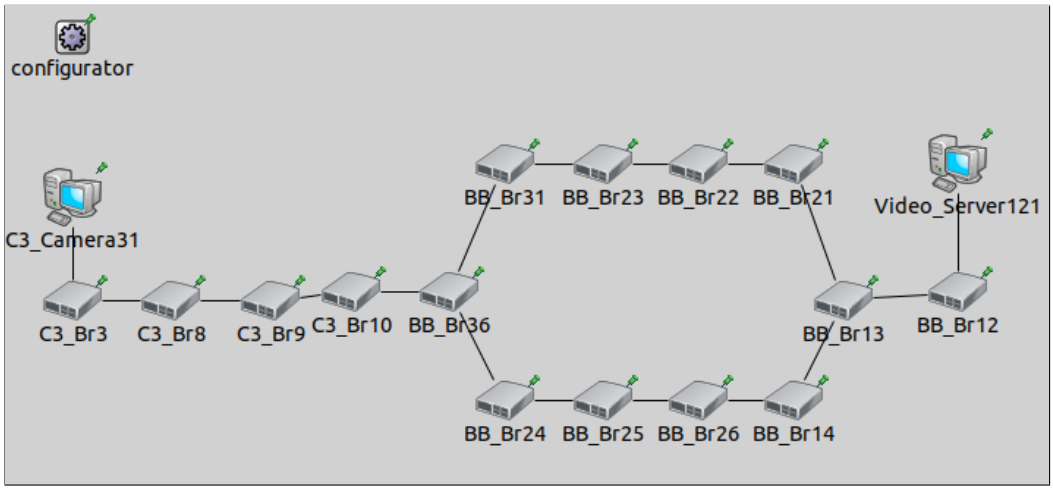

Figure 5.16: Part of the SN topology in Tsimnet

As noted in the slice definition, the link mapping algorithm used is MinBr and two paths are computed. After calculating the mapped paths in ALEVIN and exporting only the relevant portion of the SN, the configuration tables: stream identification, sequence generation, stream identity, stream splitting, and stream recovery are computed. Because of the long list of variables in some tables, the entries are presented only partially.

Firstly, the stream identity table is presented. It includes all streams in a system, and since the slice used includes only one VLi, two streams are generated. Table 5.8 shows the most important computed values of the stream identity table entries. The identification

type is taken as null stream identification since it is the required type for each network entity, and VLAN ID is the ID of the VN in ALEVIN.

Table 5.8: Stream identity table

Stream ID	Identification type	VLAN ID
21	Null_stream_identification	1
4	Null_stream_identification	1

The sequence generation table shows for which streams is a sequence generation function to be instantiated, and in which direction (whether on in- or out-facing side of the ports). The values are shown in Table 5.9. The direction infers whether the sequence generation function is in- (false) or out-facing (true). The stream with *stream_handle=21* is out-facing because it is generated at the talker-end system, whereas the stream with *stream_handle=4* is generated at the relay system.

Table 5.9: Sequence generation table

Stream list	Direction
21	True
4	False

The sequence identification table instructs the system where to instantiate the sequence encode/decode function. Table 5.10 shows which stream packets are to be encapsulated by the noted encapsulation type at which port with the direction. The boolean variable "active" shows whether it is an encode/decode (true) or only decode (false) function. Moreover, according to the standard, it is recommended to encode the packets with the R-TAG; therefore it is used as the default value for the configuration.

From the last table, we can find the nodes at which the two streams split and merge. Those are the nodes with both streams [21, 4] in the stream list. Since the entries are generated following the mapped paths, the main stream splits at the node *BB_Br36*, and the member streams meet at the node *BB_Br13*. Another information which can be obtained from the table is the listener node (*Video_Server121*) since it is the only node that has only the decode function instantiated (*active=False*).

The stream splitting table instructs the system at which ports which streams need to be split and into which member streams to split them. Table 5.11 shows that the stream with *stream_handle=21* duplicates at the *BB_Br36:1* and it is split into streams 21 and 4. According to the standard, it is allowed for at most one duplicate to have the same *stream_handle* (21). The event log from Figure 5.17 shows the splitting of the stream and forwarding of the new streams at the common node of the redundant paths *BB_Br36*.

At this point, the system has information about which streams exist, for which streams is it required to generate the *sequence_number* parameters, at which ports to encode and decode packets of which streams, and about the splitting of the streams. Next step is to instruct the system where to instantiate the recovery functions. Both recovery functions, individual and cumulative, are presented in the sequence recovery table. The values of the important variables are given in Table 5.12.

The stream list contains the list of stream handles on which the recovery functions will operate. The standard gives two options for the recovery algorithm: VectorRecovery and

Table 5.10: Sequence identification table

Stream list	Port	Direction	Active	Encaps. type
21	C3_Camera31:1	True	True	R_TAG
21	C3_Br3:1	False	True	R_TAG
21	C3_Br8:1	False	True	R_TAG
21	C3_Br9:1	False	True	R_TAG
21	C3_Br10:1	False	True	R_TAG
21, 4	BB_Br36:1	False	True	R_TAG
21	BB_Br31:1	False	True	R_TAG
21	BB_Br23:1	False	True	R_TAG
21	BB_Br22:1	False	True	R_TAG
21, 4	BB_Br13:1	False	True	R_TAG
21	BB_Br12:1	False	True	R_TAG
21	Video_Server121:1	True	False	R_TAG
4	BB_Br24:1	False	True	R_TAG
4	BB_Br25:1	False	True	R_TAG
4	BB_Br26:1	False	True	R_TAG
4	BB_Br14:1	False	True	R_TAG

Table 5.11: Stream split table

Split port list	Direction	Input stream list	Output stream list
BB_Br36:1	False	21	21, 4

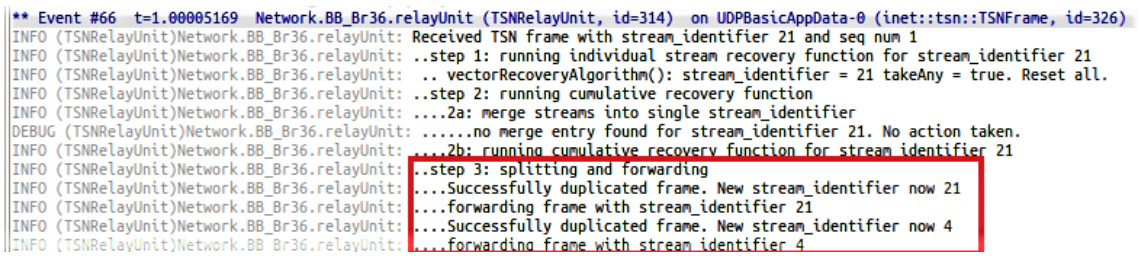

Figure 5.17: Stream splitting function instantiation at the splitting node

Table 5.12: Sequence recovery table

Stream list	Port list	Algorithm	Individual	Latent error
21	C3_Br3:1, C3_Br8:1, C3_Br9:1, C3_Br10:1, BB_Br36:1, BB_Br31:1, BB_Br23:1, BB_Br22:1, BB_Br21:1, BB_Br13:1, BB_Br12:1	Vector	True	False
21	Video_Server121:1	Vector	True	False
4	BB_Br24:1, BB_Br25:1, BB_Br26:1, BB_Br14:1, BB_Br13:2	Vector	True	False
21, 4	BB_Br13:3	Vector	False	True

MatchRecovery algorithm. Moreover, individual recovery is instantiated for both streams on all forwarding nodes of the calculated paths. The cumulative recovery is instantiated only on the *BB_Br13* because it is the merging node of the calculated paths. The latent error function is activated only with the cumulative recovery. Node *C3_Camera31* does not exist in the table since it is the origin of the stream, and a recovery function at the talker node is not required. The recovery functions are necessary because of two reasons: the individual recovery is important for faults in one member steams such as a stuck transmitter, and sequence/cumulative recovery is needed for eliminating the duplicate packets when merging the member streams.

Figure 5.18 shows the instantiation of an individual recovery function at node *BB_Br14*. The sequence number of the received frame from stream 4 is verified (delta from old seq num), then the frame is forwarded.

```
** Event #1055  t=6.000094194  Network.BB_Br14.relayUnit (TSNRelayUnit, id=821)  on UDPBasicAppData-5 (inet::tsn::TSNFrame, id=4211)
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: Received TSN frame with stream identifier 4 and seq num 6
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ..step 1: running individual stream recovery function for stream_identifier 4
DEBUG (TSNRelayUnit)Network.BB_Br14.relayUnit:  .. vectorRecoveryAlgorithm(): stream_identifier = 4 has a delta of 1 from old seq num 5
DEBUG (TSNRelayUnit)Network.BB_Br14.relayUnit:  .. vectorRecoveryAlgorithm(): seqNumBitmask before 1 after 1.
DEBUG (TSNRelayUnit)Network.BB_Br14.relayUnit:  .. vectorRecoveryAlgorithm(): normal in-order case here. New mask 1 with lastSeen = 6
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ..step 2: running cumulative recovery function
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ....2a: merge streams into single stream_identifier
DEBUG (TSNRelayUnit)Network.BB_Br14.relayUnit: ......no merge entry found for stream_identifier 4. No action taken.
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ....2b: running cumulative recovery function for stream_identifier 4
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ..step 3: splitting and forwarding
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ....no split match. No action taken.
INFO (TSNRelayUnit)Network.BB_Br14.relayUnit: ..step 4: forwarding stream identifier 4
```

Figure 5.18: Individual recovery function instantiation - packet forwarding

In order to enable the sequence recovery function on the compound stream, a merge table is added to the simulator. It includes lists of streams to be merged and to which stream_handle. As shown in the event log in Figure 5.19, stream 4 is merged with stream 21 in node *BB_Br13*, and the frames are further forwarded with the *stream_handle=21*. All packets coming at the node *BB_Br13* with *stream_handle=4* will be overwritten with the new stream_handle value. In this case, the individual recovery function at node *BB_Br13* will treat the framed received with *stream_handle=4* as duplicates and will drop them as seen in figure 5.20.

The sequence recovery function can be instantiated with or without a latent error detection function. In general, the computed configurations imply a latent error detection

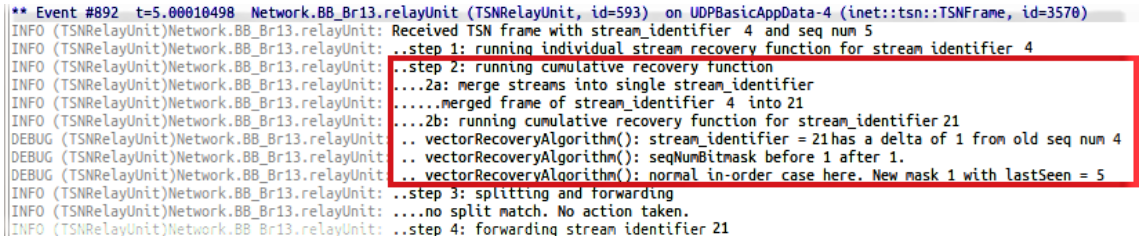

Figure 5.19: Sequence recovery function - merging two streams

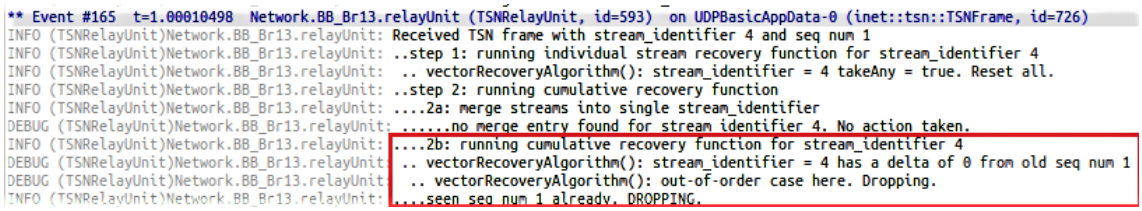

Figure 5.20: Sequence recovery function - elimination of duplicate packets

function in the case of a cumulative recovery function since the latent error detection provides additional reliability by counting the discarded packets. In case there are less discarded packets than expected, it means there is a failure in one of the redundant paths. TSimNet simulator is not a complete implementation of the FRER and one of the missing components is the latent error detection function, and therefore, its instantiation is not possible.

The validation of our configuration tables in the simulator is performed partially. In spite of its incompleteness, TSimNet is the best option for simulating FRER at the time of this work. The tested configuration tables in the simulator are the sequence recovery table and the stream split table. However, all configuration tables have been assessed for conformance to the standard.

5.8 CONCLUSION

In this chapter, we presented two link mapping algorithms with different levels of redundancy. MaxBr calculates a redundant path with a minimal number of common links with the main path. MinBr calculates a minimally disjoint backup path (branch) that can satisfy the reliability demand. The calculation of the shortest paths is performed by a traditional k-shortest paths algorithm, with reliability as a weight (RSP). The algorithms work under the assumption that the reliability of each entity of the physical topology is known. The evaluation of our work showed the necessity of using different algorithms according to the reliability requirements of the traffic classes and the topology. We showed a trade-off between resource utilization of the SN, and the achieved reliability and acceptance ratio of the slices.

Furthermore, in order to make our results applicable to future industrial networks, the embedding results were integrated with the TSN standard IEEE 802.1CB FRER. The advantages of FRER components in a network are numerous, as discussed by researchers and companies in the networking field. Therefore, the integration of our embedding results can contribute to more efficient use of TSN networks that incorporate the FRER capabilities. For validating the computed network configurations, TSimNet simulator

has been used and some adaptations were required in order to import and apply the configurations properly.

Using this work, the mapping results of VNE algorithms, with the assignment of reliability values to network components, can be integrated with the FRER with a reasonable effort, since the network configurations are generated automatically. Future work might focus on extending TSimNet to apply all network configuration tables in accordance with FRER. The complete applicability of branching in the combined solution (EVN) requires virtualizing FRER in a similar approach to virtualizing TAS. The mapping results from ALEVIN are already the configuration method of such a virtual FRER.

6 SECURITY PERSPECTIVE

Two significant challenges in NFVI are how to achieve a comprehensive and autonomic Service-level Agreement (SLA) management, and how to flexibly deploy and activate security mechanisms on-demand. The main research problem in this chapter is autonomic security management in NFVI. The main goal is early reaction to threats through SFC reconfiguration via VNF live migration. This goal is approached by supporting the security measurements with a decision making architecture that considers, on the one hand, the threats and events in the environment, and on the other hand, the SLA between the communication service provider and customer. We design a decision engine and identify the significant SLA metrics and relevant SFC placement policies. We realize the security-aware placement policies using VNE algorithms. Furthermore, we analyze the VM-specific attacks and define possible early detectable behavior patterns [7]♮. Since we chose the VM technology to realize the VNF, we consider that the VM-specific threat analysis applies to the VNF. These threats mainly exist in public clouds that might host VNFs at the cloud level in our main use case. These VNFs might process critical data or decisions for the industrial enterprise.

6.1 VM-SPECIFIC ATTACKS

This section is an extension of sections "Co-location", "Attacks on migrated VMs", "Cross-VM side-channel attacks", "Exploiting live migration" in the author's publication [7]♮.

A literature survey about attack vectors, attack behavior, defense measurements, and technical reports about attacks has been conducted, and a first behavior pattern has been identified [7]♮. Typical VM-specific attacks are cross-VM cache-based side-channel attacks. In these attacks, the attacker needs first to gain and verify co-location with the victim VM. The methods of forcing and verifying co-location with a victim VM use VM instance types, availability zones, IP addresses, and Internet Control Message Protocol (ICMP). Those methods expect the behavior of the VM placement algorithms and also use side-channels. For example, by applying a varying load on the victim VM and measuring the memory access time of the attacker's VM. The noise of other VMs and the host are challenges to these attacks. An example of defense measurements is VM location obfuscation using user-defined or dynamic VM placement algorithms.

A typical cache-based side-channel attack is Prime and Probe attack. The attack fills the cache then measures the memory access time. If the access time of a certain memory address is higher than a threshold, a memory page associated with the respective cache line was accessed by the victim. A researcher demonstrated a Prime and Probe attack used to recover a full 2048 bit RSA key in Amazon EC2 cloud [167]. Another cache-based side-channel attack is Flush and Reload attack [178]. The attack relies on memory deduplication feature (sharing common system libraries in the memory). It flushes certain known library memory lines from the cache, waits, then measures the access time. Frequent flushing of the cache is a candidate behavior pattern of such an attack.

6.1.1 *Cross-VM Side-Channel Attacks*

A security-critical property in NFVI is the isolation between co-residing VNFs of different customers. Co-hosting creates an attack surface that is impossible or at least harder to exploit in a non-virtualized environment. In traditional external side-channel attacks, the attacker tries to gain information about the victim from the network traffic, power consumption, response time, etc. In virtual environments, malicious VMs can perform sophisticated side-channel attacks by monitoring the computing resources in the host. However, this requires the attacker to place his VM in the same host as the victim VM. Most of the research about cross-VM side-channel attacks focuses on memory- and cache-based side-channels, since memory and cache are shared among VMs of costumers and hold easily accessible run-time data of VMs. The known attacks are discussed in the following paragraphs. We present an overview of cache- and memory-based side-channel attacks discussed in the literature and the main mitigation measurements proposed to hinder these attacks or make them difficult.

PRIME AND PROBE ATTACK Prime and probe attack operates in three steps. In the Prime phase, the attacker fills a portion of the cache with his data. In the Trigger phase, the attacker waits for the victim to access the memory. In the Probe phase, the attacker measures the time of each memory access he makes. If this is higher than a threshold, the accessed memory page is not cached anymore. Therefore, a memory page associated with this cache line was accessed by the victim (or other VMs). The first experimental illustration of cache-based side-channel attacks has been presented in [167]. Such attacks has been shown to recover a full 2048 bit Rivest–Shamir–Adleman (RSA) key in a vulnerable cloud environment [167]. To perform the attack efficiently, the authors analyzed the last-level cache structure of the Intel Xeon E5-2670 v2 CPU, which was dominant on Amazon EC2 cloud servers for the target instance types. The authors adapted the Prime and Probe attack presented in [168] and expanded it to show its applicability to *Libgcrypt* implementation of RSA. For the key recovery, the authors used the Prime and Probe attack to leak the accessed positions of the table that stores the decrypted ciphertext, and synchronized the decryption with the spy process by requesting decryptions using Transport Layer Security (TLS) requests.

The authors in [169] presented a cache-based side-channel attack to extract SSH keystrokes from another VM. They have been able to observe the keystrokes sent with 5% missed keystrokes and 0.3 false triggers per second and an observation resolution of $13ms$. The authors adapted a network-based SSH keystroke timing attack presented in [170]. This attack uses timing differences between letters typed by different fingers and hands to determine what the user is typing. The original attack uses a network tap to monitor the inter-packet timing and assumes every packet sent to the SSH server to be a keystroke. The authors generated a map of timings between certain character pairs and divided them into different categories according to the hands and fingers used. The authors used a training data-set to train the algorithm. This data-set should not be necessarily created by a victim. The experiments discovered certain human behavior patterns in typing passwords and reached a significant reduction in the password search space. As a counter measurement, they mentioned introducing dummy traffic between the server and the client to introduce additional noise to the cache.

The authors of [169] adapted the original attack to exploit cache usage and timing information gathered by the Prime and Probe attack. The time measurement is done by repeatedly running the Prime and Probe attack and observing how long a specific cache usage pattern resides in the cache. The authors assume that every keystroke processed by the victim results in a cache usage spike. To simplify, they performed the attack on a local testbed to ensures that the attacker and the victim co-reside on a single CPU core and that the system is idle. To apply the attack in real environments, it had to be optimized to be more robust against noise introduced by the system and other applications running on the machine, and against core migration (switching CPU core for one process or VM). Another challenge for this attack is that the noise introduced by the response messages from the SSH server has to be detected and filtered. The authors filter cache activities according to a certain duration range.

CRYPTOGRAPHIC LIBRARY CACHE TIMING ATTACK The authors of [171] used cache timings of Advanced Encryption Standard (AES) operations performed by widely used cryptographic libraries. The attack is divided into four stages. First, the attacker creates a profiling server that uses the same software and hardware as its victim. The attacker starts encrypting the plaintext using a known key and stores the timing information for each byte of the key into a table. In the attack stage, the attacker sends a known plaintext to it's victim and stores timing information just like in the profiling stage. Afterward, the attacker correlates the timings measured in the first two stages and stores the correlation coefficients for every possible key byte in a new table. Finally, the attacker brute forces all possible key combinations using the correlation table.

To blindfold this attack, the authors suggest using available hardware extensions such as the AES-NI, which is available in most modern CPUs. The AES-NI performs a full round of AES without accessing cache or memory in an atomic operation. This makes timing and other cache-based side-channel attacks useless against AES, as the computed data is not fetched to the cache and the computation can't be intercepted by an attacker. However, it is impossible to provide special instruction sets for every cryptographic algorithm. If such instruction sets are unavailable, the authors suggest using cache prefetching, in which all necessary information for the computation are fetched to the cache before starting the computation. Prefetching has to be repeated every time the attacking VM runs a Prime phase. This can be done by the VM monitor by refilling the CPU cache after each context switch between residing VMs. This process is called cache warming.

FLUSH/RELOAD ATTACK. Memory deduplication is a memory utilization optimization technique introduced in [172]. It seeks memory pages with the same content and merges them into one physical memory page. Merged memory pages are tagged as read-only to prevent one owner from modifying them. If the page needs to be modified, its modified version is stored in another memory page. Depending on the special implementation, not all memory pages can be deduplicated. The Linux implementation, for example, only deduplicates memory areas that are allowed to be shared by a system call [173]. This approach has been later adapted and modified to get rid of the client-side modifications needed and implemented for VMWare ESX VM monitor in [174]. A similar method has been introduced in [173] for Linux kernel version 2.6.32, and hence available for KVM. For XEN, two approaches have been presented in [175] and [176].

As deduplication is used for shared libraries, the attacker can access and cache the memory pages used by the victim and use an attack technique called Flush/Reload. This technique has been first described in [177] and later named in [178]. The Flush/Reload attack also has three stages. In the Flush stage, the attacker flushes the CPU cache lines that might be accessed by the victim. This can be done using the `clflush` instruction that flushes the CPU cache line for the given virtual memory address. The addresses can be identified, for example, for a certain shared library that the victim needs to access to provide a certain service. The `clflush` instruction also flushes the CPU cache lines in all cache levels across all cores. In the Trigger stage, the attacker waits for some time to give the victim the chance to access the memory portion. In the reload stage, the attacker accesses the previously flushed memory addresses and measures the time needed for this to figure out if the victim accessed these addresses or not.

As the *clflush* instruction flushes the cache lines from all cache levels, the attack can use the last-level cache to work across the CPU cores. An attack that exploits memory deduplication is presented in [179]. The authors stated that this attack can be used for all block ciphers that use a table lookup followed by a key addition. The demonstrated attack on AES works as follows: first, the attacker needs to get the offset of the tables for the last round with respect to the first address of the library. With these offsets, the attacker can access each memory line of the tables even if address space layout randomization is enabled. Afterward, the attacker requests encryptions from the victim and uses the Flush/Reload technique to determine the table which was accessed. This information is stored with the ciphertext. In order to recover the full key and to reduce noise, these measurements are repeated several times. Finally, the attacker can recover the encryption key using his measurements and the knowledge about the public tables. The experiments show that a full AES key can be recovered by observing 2^{19} encryptions in the cross-VM testbed.

6.1.2 *Co-location*

Co-location with the victim VM in the same host is a prerequisite for shared resource cross-VM side-channel attacks. In general, an attacker might exploit the weaknesses in VM placement algorithms and the lack of location privacy in cloud environments to gain and verify co-location. Using network-based methods, internal IP addresses assigned to instances can be mapped to availability zones and instance types, as described by the case for Amazon's EC2 in [169]. The internal IP was easily resolved by looking up the Hostname of a VM using an EC2 internal DNS resolver. As the internal IP depends on availability zone and instance type, the attacker is able to use this map to make an educated guess about the parameters of instance creation to raise the probability of gaining co-location by flooding instances of the determined type and availability zone. This method can be mitigated by assigning randomized internal IP's to each VM independently from availability zones and instance types, or by restricting the visibility of internal IP's. Furthermore, Amazon provides information on the availability zone and their current IP address ranges publicly as a JSON file. Another method to check co-location is measuring the round-trip-time. The round-trip-time should be lower if the attacker's VM and its victim's VM co-reside. This approach is hard to defeat as low response times are highly important for most services on the Internet.

According to [169], each physical machine is assigned to one or two instance types, and each machine can support a specific number of VMs of this type. This makes it easier for an attacker to gain co-location as it reduces the number of physical machines a specific VM of a known instance type and availability zone may reside on. This makes the brute-force-based placement of malicious instances cheaper. Additionally, it is not highly probable that two instances of one customer are placed on a single physical machine. This is intended to avoid a single point of failure for the customer and balance the load over different physical machines. For an attacker, this strategy has the advantage that the created malicious instances will probably not co-reside, which raises the probability of gaining co-location to a specific VM. On the other side, the attacker can't isolate the victim by filling the hosting physical machine with his own malicious instances. This introduces noise to the measurements by other VMs that might not be important for the attacker. The attacker can also use the parallel placement locality. This means if two or more instances are created from different customer accounts shortly after each other, it is more likely that these instances co-reside. The authors show that this short time can be more than an hour between the creation of the two instances. An attacker can wait until the victim creates a new instance or can try to force the victim to create new instances by introducing load to the victim's service if the victim automatically creates instances to balance the load.

As a counter measurement, the authors propose that each customer should decide about the placement of his own VM. This counter measurement is taken by Amazon using the virtual private cloud, which provides all the benefits of Amazon's cloud API on single-tenant hardware. The authors of [180] and [181] introduced new placement algorithms that are more robust against these attacks. In [180], the algorithm marks all servers as 'on' or 'off'. Additionally, all 'on' marked servers are marked as 'open' or 'closed', indicating if a server can still accept new VMs. The controller instance always keeps λ servers online ('on'), which can still host new VMs ('open'). Every newly created VM is assigned at random to one of these 'open' servers. In [181], all physical servers are assigned to a group of an adjustable size. A new group of servers is only started if all running groups of servers can't run a newly created VM. The algorithm prefers to place newly created instances in servers on which the customer's VM resides on or resided on in the past. Finally, if a new customer never created a VM before, the newly created VM is assigned to the server with the least number of VMs from all available groups of servers.

The authors of [169] also presented a modified and simplified cache-Based side-channel method to verify co-location. First, the attacker loads a buffer into the cache, the so-called Prime phase. In their experiments, the authors use a buffer of a size that fills a significant portion of the cache. Afterward, the attacker's spy process waits some time to give the victim the chance to access the memory. Finally, the attacker measures the time (in number of CPU cycles) needed to access the buffer, the so-called Probe phase. If the victim was active in between, the access time is considerably higher since the victim accessed the memory and therefore replaced the cache lines. This method requires knowledge about the load on the victim VM or the services provided by it. The attacker tries to introduce varying loads to the victim VM and identify the variation in the cache hit/miss rate. The authors used a simple Apache 2.0 web server serving one 1024 byte text-only HTML file as a victim. The attacker introduces varying loads to the server using HTTP get requests. If both VMs co-reside, the load should result in higher memory access times

(higher cache miss rate) in the Probe phase. The attacker might also profile the services provided by the target, such as en-/decryption.

By observing cache usage patterns, the authors of [182] were able to verify co-location with a certainty of 50% in the worst case and up to 90% in the best case. The authors also introduce memory bus locking to degrade the server's memory performance to verify co-location. Memory bus locking uses an atomic operation that forces the CPU to flush all running memory transactions. To increase the effect, the attacker needs to choose an instruction with a long execution time. As the server's memory performance is degraded, the performance of all residing VMs is degraded depending on their memory usage. The authors show a performance degradation with a factor higher than 4 for an Apache web server. As the performance is degraded, the service can't handle many requests. An attacker can then verify co-location by profiling the performance of the target.

Another method that could be used to verify co-location is presented in [183] using energy consumption. The authors use a metered rack power distribution unit, a common hardware in data centers to monitor the power consumption, that they access using Simple Network Management Protocol (SNMP). The attacker can introduce varying load to the target VM to identify the server on which the target resides. The varying load should be visible through correlating a varying power consumption. The authors showed that the power consumption profile is almost independent of the used CPU and mostly depends on the service provided by a specific VM. However, this method is sensitive to noise produced by other VMs that can blindfold the detection of a VM with a low power consumption profile.

6.1.3 *Exploiting VM Migration*

One of the main features of IaaS clouds is the live migration of VMs in which a running VM is moved to another server with the least possible interruption. Live migration improves the flexibility of the virtual environment and allows the provider to keep the VM running with the required performance when the original host is overloaded or has to be isolated for maintenance or because of an error or attack. Live migration can also be utilized for improving the OPEX (e.g., energy) by consolidating VMs. The security challenges in the cloud are more serious when migration is used, in particular, if migration is performed between servers of different widely distributed data centers [184]. Two main threats are imposed by live migration: the exploitation of the migration itself, and attacks on the customer VMs during migration. In the second threat, the migrated VMs might face different attacks such as man-in-the-middle, denial-of-service, and stack over-flow [185]. These attacks can be either active attacks that change the migrated data or passive attacks that perform eavesdropping on the VM to extract sensitive data such as passwords [186]. The migration data such as kernel memory, application state, sensitive data such as passwords and keys, can be sniffed or tampered easily if transmitted without encryption, thus compromising the integrity and confidentiality of the VM data [185].

6.2 SLAS AND DEPLOYMENT POLICIES

Public NFVIs enable flexible resource sharing among customers but open a wide surface of threats on customer's data from both the provider and co-hosted customers. Quality

of Service (QoS), Privacy, and Security (QPS) are main challenges in public NFVIs that might be conflicting. For example, privacy is threatened when the provider applies certain security measurements to protect the environment and other customers. The NFVI provider needs placement policies for SFCs and security mechanisms to conform to SLAs, avoid financial and reputation losses, protect the environment, and maximize the revenue. However, a comprehensive view that covers QPS issues through SLAs and placement policies, mainly for SFCs, is still missing. In this chapter, we research the existing SLA metrics and placement policies in public IaaS and define new metrics and policies that consider the SFCs.

SLA is a formal contract between the service provider and a subscriber that contains detailed specifications called service level specifications in order to guarantee a certain QoS [187]. The SLAs are contractually binding and the service provider strives to prevent SLA violations to enhance customer satisfaction and to avoid penalty payments [188]. The dependencies between QPS issues need deep investigation to define the best forms of SLAs that satisfy both the NFVI provider and customer. In this section, we describe essential QPS requirements for both NFVI provider and customer who deploys a SFC. Furthermore, we propose the placement policies that the provider should apply to fulfill these requirements. The QoS SLA metrics and deployment policies discussed here are mainly related to security measurements. However, we mention the main policies from the previous chapters in the context of a comprehensive SLA. Furthermore, we do not implement all these metrics and policies in our approaches.

SLA METRICS The provider and customer need to agree on five types of metrics related to QPS, financial issues, and exceptional measurements. In the following, we list these types and target metrics:

- QoS SLA metrics cover the customer's service availability and responsiveness:
  - VNF availability: can be interpreted as the total downtime (interruption) within a certain period. This downtime might result, for example, from live migration of a running VNF to another host for many reasons such as maintenance and suspicious behavior. An example of this metric is an availability of 99.99% that allows a maximum downtime of 53 minutes per year.
  - SFC availability: related to the downtime of any VNF of a linear sub-chain within a certain period.
  - SFC response time.
- Protection of customer data from other customers:
  - Service co-location with another customer: this metric specifies whether the VNFs are allowed to be co-hosted with VNFs belonging to other customers on the same host.
  - Availability of certain security mechanisms such as Virtual Machine Introspection (VMI) and trusted hardware in the host.
- Privacy of customer data against the provider itself:
  - Restricting the processing of customer data in certain geographical locations. For example, processing the German financial data only in Germany.

  - Full scan of the VNF by security mechanisms such as anti-virus programs.
  - Tracing/logging VNF activities, for example, by hypercall-tracing that monitors all VNF activities in the virtualized hardware.

- Financial commitments of both the cloud provider and customer:
  - The price of hosting the customer service. This price is usually determined by the reserved computing resources and the time period (e.g., Amazon EC2).
  - Financial penalty for the provider under the violation of each SLA metric. In general, the violation of data privacy and security metrics shall impose an absolute penalty. On the other hand, the violation of QoS metrics shall impose a penalty relative to the measured service degradation. For example, the provider shall return a certain rate from the service price for each minute of service downtime that exceeds the availability threshold defined in the respective metric.
  - Financial penalty when the VNF causes damage to the environment. This penalty should also be fixed for a severe damage, or relative to a measured damage such as host performance degradation for a certain period of time. However, in this case, there shall be a clear evidence on the malicious behavior of the VNF and the source of infection.

- Exceptional conditions that dominate any other metric. These metrics define the conditions under which the provider can violate a certain metric. For example, a suspicious behavior by a VNF, or a direct damage to the environment such as performance degradation in Denial of Service (DoS) attacks. The SLA should define certain rights for the provider in such cases. For example, moving the VNF to a more protected environment to perform full scanning, even if this will violate a privacy or a QoS metric.

DEPLOYMENT POLICIES After defining the SLA metrics for all five aspects (QPS, financial, and exceptions), the NFVI provider shall define the policies used to place the customer's SFCs in its hosts. These policies shall be mapped to respective SLA metrics, and managed by SFC deployment algorithms that force this mapping and maximize the provider's revenue. We define QPS policies that satisfy the customer requirements, and provider-specific policies to increase the revenue and minimize the financial penalties and losses. The policies might be either applied by the deployment algorithm at the first instantiating of the SFC, or applied on-demand as a reaction to certain events in the environment or the SFC. Some of these policies are applied by the EVN composition and embedding algorithms presented in Chapter 3. The on-demand based application of policies also needs mechanisms to monitor the behavior and performance of deployed SFCs. In the following, we list a set of proposed policies for each type and map each policy to a certain SLA metric.

- QoS policies (customer-oriented):
  - Instantiating service VNFs in nodes with certain computational capacities to satisfy service response time metrics.
  - Instantiating service VNFs in customer dedicated hosts to satisfy service response time metrics.

  - Instantiating service VNFs in certain geographical locations (e.g., edge) where many requests come from to achieve the required response time.
  - Instantiating backup SFCs permanently (branching) or during downtime.
  - Placing service VNFs such that the required communication delay in not violated. For example, by placing a TAS VNF on each host to schedule the traffic.
  - Instantiating service VNFs in different hosts to force load balancing among hosts and improve the performance. Such a policy is applied in our chain embedding heuristic in Chapter 3.

- Security policies (customer-oriented):
  - Instantiating service VNFs in customer dedicated nodes.
  - Instantiating service VNFs in nodes that provide the required security mechanisms.
  - Adding certain security VNFs to the SFC.

- Privacy policies (customer-oriented):
  - Avoid tracing or scanning service VNFs.
  - Instantiating service VNFs only in certain geographical locations.

  Provider policies:
  - Scanning/tracing a VNF/SFC under exceptional conditions such as suspicious behavior.
  - Stopping a VNF/SFC under confirmed attacks.
  - Prioritizing VNFs/SFCs according to the revenue from hosting customer's services and penalties imposed by SLA violation for this customer. These priorities might help the deployment algorithms when responding to certain events. For example, if a certain VNF with high penalty for service response time violation, is co-hosted with other customer VNFs, and goes through a performance degradation, it shall be prioritized over the other VNFs. In this case, the other VNFs might be migrated to other hosts.

SCENARIO Figure 6.1 depicts a practical scenario that combines a set of the discussed QPS SLA metrics and policies. The figure also presents a set of assumed QPS monitoring and management components that are needed to help the NFVI provider to force the policies that protect the environment and at the same time satisfy the SLAs. A typical data center tree topology is used with two types of nodes. The first type is production nodes that host customer VNFs and VMI to perform a lightweight monitoring of the VNFs. The second type of nodes is protected nodes (analysis nodes) that run additional protection components such as anti-virus programs. A QPS management node receives events from VMI, the VNFs placement status and nodes types from the NFVI management system, and the network topology from a network management system. The QPS management node shall also be aware of the deployed SFCs with their SLAs and the respective placement policies. The customer deploys an SFC with three VNFs. The SLA of this SFC has the following metrics:

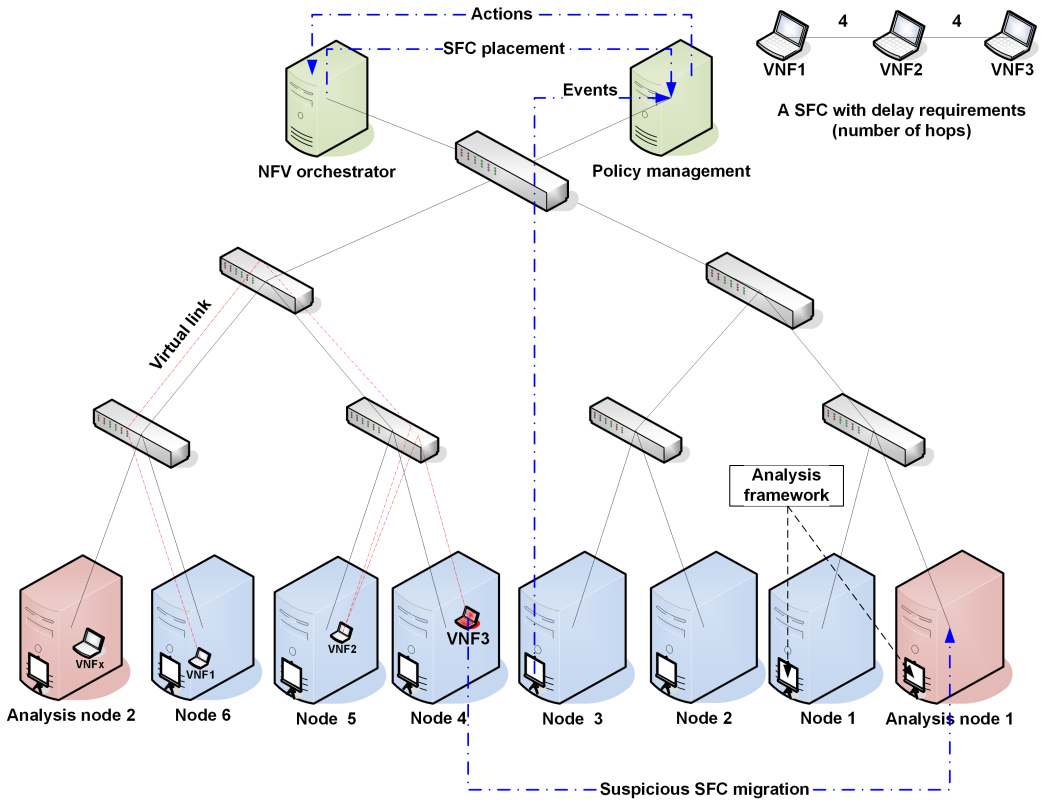

Figure 6.1: An example of the proposed SFC deployment policies

- The SFC has a latency constraint of 4 hobs between each two adjacent VNFs (for simplicity).
- The availably of the service is 99,99%. This means the service cannot be down for more than 53 minutes over a year.
- No co-hosting of two VNFs in the same node.
- The VNFs cannot be co-hosted with another customer.
- The VNFs cannot be fully scanned under normal conditions; this means that they cannot be hosted in a fully protected environment.
- The exception for the last metric is when the VNFs show a suspicious behavior that might threaten the environment.

According to this SLA, if a certain VNF from the customer service shows a suspicious behavior, the QPS management can migrate it to a protected node. The migration should consider the other constraints. For example, a protected node that hosts VNFs for other customers cannot be chosen. Another example is when the nearest protected node will violate the latency constraints. In this case, the whole SFC needs to be replaced.

However, many challenges arise from this research domain and still need deep investigation:

- Collecting evidences about suspicious behavior by the customer's service VNFs.

- How the customer can check the provider's compliance with security and privacy SLA metrics?
- How the conflict between customer's privacy and the security of other customers and the cloud provider can be balanced?

6.3 DECISION ENGINE

This section is an extension of Sections 3.3 of the author's publication [4]↯, mainly with the prototype description.

The decision engine in Figure 6.2 is a component and a central coordinator in a complete malware defense architecture presented in [4]↯. This decision engine reacts to a possible indicator of an attack by initiating certain actions according to the expected attack, SLAs, and policies defined by the provider. The most significant action is migrating a suspicious VNF to a dedicated analysis environment, to protect the production environment from the suspicious VNF and avoid overloading it with heavy analysis mechanisms. In this case, a full SFC reconfiguration might be required to fulfill the customer's SLA. The interfaces, inputs, and outputs of the decision engine are defined.

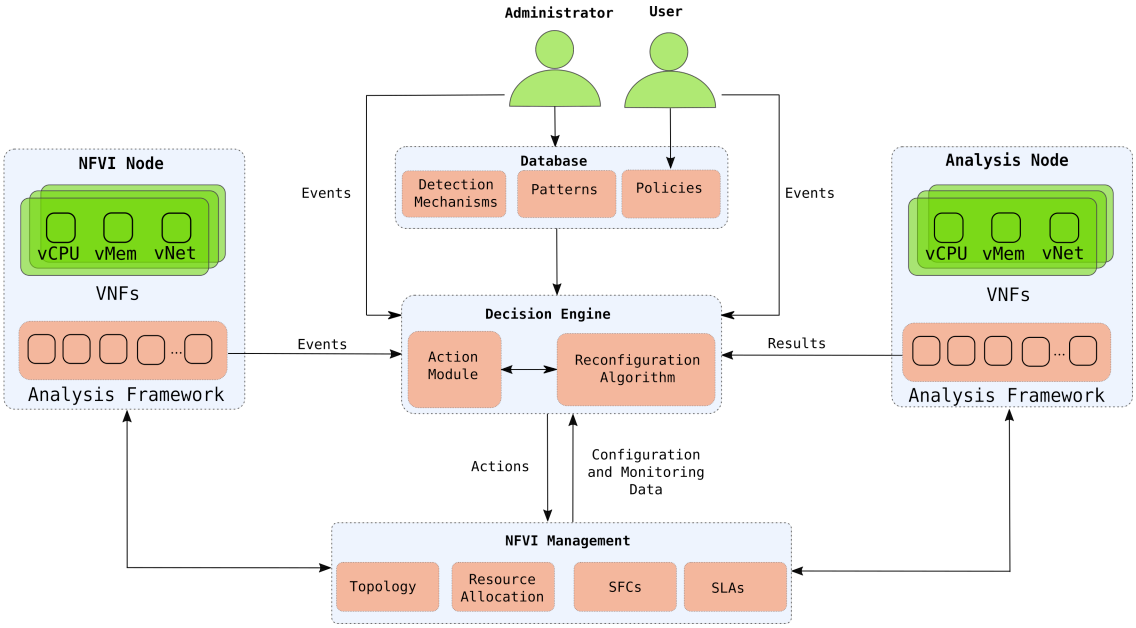

Figure 6.2: Decision engine structure[4]↯

The engine receives events from an assumed NFVI monitoring system that deploys lightweight tracing mechanisms to monitor VNF activities in the production environment. The events represent certain suspected attacks based on predefined suspicious behavior patterns. The engine reacts to events according to user-defined policies. The event defines the suspected VNF and suspected attack/malware. We assume that the provider needs individual policies for each VNF according to its SLA. The policy defines a set of reactions to a predefined event. The main threat reaction under focus in this work is isolating a suspicious VNF in a dedicated analysis environment (protected host) to protect the production environment from attacks and performance degradation. The engine uses a

migration algorithm to select the protected host that does not host any VNF and offers enough computing resources to host the target VNF.

The distribution of specialized investigation hosts plays a main role in the decision process. Finding an efficient distribution of these hosts is also an economic challenge of the proposed architecture. This requires a detailed analysis of the target environments and expected attack scenarios. The trade-off between protection costs and attack costs will be a major output of this analysis. Another economic aspect is the minimization of the VNF's downtime during the migration to prevent a monetary [189] and a reputation loss [190] for the provider. The service provider commits to a certain level of service, which is described by an SLA.

Usually, the SLA expresses the VNF availability. For example, 99.999% guaranteed availability means 5 minutes of downtime per year. Non-compliance to such SLAs can lead to (monetary) penalties for the providers and can harm their reputation [191]. Reputational damage leads to economic long-term consequences, because the consumer's trust in the service might be lost and fewer customers might use the provider's services in the future. Therefore, the downtime and migration time of the VNF need to be minimized. Although live migration usually causes only a short downtime to a VNF, small interruptions ranging from 60 milliseconds to 3 seconds are inevitable [192]. In order to guarantee a certain QoS it is, therefore, essential to predict the worst-case downtime as precisely as possible [193].

6.3.1 *Inputs*

The decision engine depicted in Figure 6.2 is the central component of the proposed architecture. It has communication interfaces with the NFVI management system, detection mechanisms deployed in the nodes, a set of databases, and the NFVI administrator and customer. The main role of this engine is to react to alarms received from the detection mechanisms when certain behavior patterns are detected. The reaction can be a certain reconfiguration action in the cloud environment, or raising an alarm to the administrator or customer. To perform this task, the decision engine needs four main inputs.

The first input is the configuration and monitoring data acquired from the NFVI management system and contains four data sets. The physical network defines the NFVI hosts and analysis nodes and detection mechanisms deployed in them, the resource capacities and usage, and the network topology. The SFCs deployed in the environment define their topology, resource demands, and constraints. The resource allocation defines how resources are allocated to the SFCs. The SLAs define the QPS requirements of the SFCs.

The second input comes from three databases. The *behavior patterns* database is maintained by the administrator and by the analysis nodes that update the information about possible attacks based on the analysis results. This database includes predefined behavior patterns that refer to a probability of a certain attack. This database consists of records of the form:

{Pattern, parameter ranges, suspected malware, possible attacks, suspicion level}

An exemplary pattern represents the memory usage spike for more than 5 seconds by the Kelihos malware that performs a DDOS attack:

{memory usage spike, period > 5 seconds, Kelihos, DoS, 50%}

The *policy* database is maintained by both the administrator and customer. It specifies a list of actions that should be executed by the decision engine when an event from a detection mechanism or an analysis result from an analysis node is received for a specified VNF. This database consists of records of the form:

{Pattern, VNF ID, Actions}

An example of such a record is:

{memory usage spike, VNF 100, block}

The *detection mechanisms* database is maintained by the cloud administrator. It is used by the decision engine mainly to find the estimated resource requirements of each mechanism. The structure of this database can only be determined after an extensive evaluation of the resource utilization by the required detection mechanisms.

The third input of the decision engine is the events received from detection mechanisms. The detection mechanisms running in nodes should have access to the behavior patterns database. When a certain behavior is detected, an event is reported to the decision engine. The event includes the following information:

{VNF ID, node ID, pattern, parameters}

An example of such an event is:

{VNF 100, Node 1, memory usage spike, period:5 seconds}

The last input is the analysis results received from analysis nodes. When a suspicious VNF is migrated to an analysis node, the node sends the analysis results to the decision engine that executes further actions defined in the policy.

6.3.2 *Actions*

The decision engine includes an action module that receives the events from the detection mechanism, reads the behavior and policy database, and determines the required list of actions to respond to this event. Possible simple actions are:

- Blocking the network connections of a VNF when the VNF is suspected to be performing a network attack.
- Restarting a VNF when facing a transient attack.
- Shutting down the VNF immediately and taking a snapshot for later analysis/prove when an attack is highly probable and the damage that might be caused to the environment cannot be easily recovered. This is required when the infection is probably propagated to the virtual disk and it cannot be recovered. This is the case, for example, when a malicious VNF is deployed in the virtual environment.

- Activating additional detection mechanisms in the cloud node to perform a more detailed analysis on the VNF if the required resources for running these mechanisms are available in the node. The decision engine reads the information about the target mechanism from the detection mechanisms database to estimate the possible overhead. The resource capacity and the current resource allocation in the target cloud node should also be read from the cloud management system and considered before this action is performed.
- The most important action in the decision engine is migrating the VNF to a dedicated analysis node where an extensive analysis can be performed without interrupting the functionality of this VNF.
- A similar action is cloning the VNF and moving the copy to an analysis node when service interruption is not feasible and the suspicion level is low. One important action after receiving an analysis result of a migrated or cloned VNF is recovering the VNF in case the analysis identifies the VNF as harmless.

6.3.3 *Reconfiguration Algorithm*

To perform the cloning and migration actions, the decision engine uses a reconfiguration algorithm that reads the configuration and monitoring data for the affected SFC and finds the available analysis nodes. In the cloning action, the algorithm has only to find an analysis node that has enough resource capacities to host the VNF and perform a detailed analysis. In the migration action, the algorithm is more complex, and three main factors should be considered in the decision making.

The first factor is the migration downtime. The live migration of VNFs usually causes a small downtime. However, a maximum downtime should be estimated before the migration is performed. The migration downtime mainly depends on the memory usage of the VNF, the network bandwidth, and migration strategy. This downtime should be then compared with the downtime budget of the VNF. The second factor is the resource capacities of the analysis nodes that have to match the requirements of the VNF. The last factor is VNFs dependencies in the SFC. The dependencies might require a full or partial reconfiguration of the SFC when a VNF is migrated to an analysis node. The reconfiguration might need to migrate other VNFs to keep the constraints of the SFC satisfied. The migration downtime and resource requirements of these VNFs should also be considered.

The decision engine waits for the events or analysis results from the production environment and the analysis environment, respectively. On one hand, when an event is received, the appropriate action will be performed according to the event parameters, detected behavior, and security policies. The migration action needs to adhere to the service requirements of the VNF. It might also require a reconfiguration of the SFC according to its dependencies. On the other hand, when an analysis result is received, the appropriate action is performed depending on the VNF state by either recovering the original configuration of the SFC if no attack is identified, or performing a certain action according to the policies if an attack is identified. An overview of the decision engine algorithm is depicted in Figure 6.3. The simple actions are blocking, restarting, and shutting down a VNF.

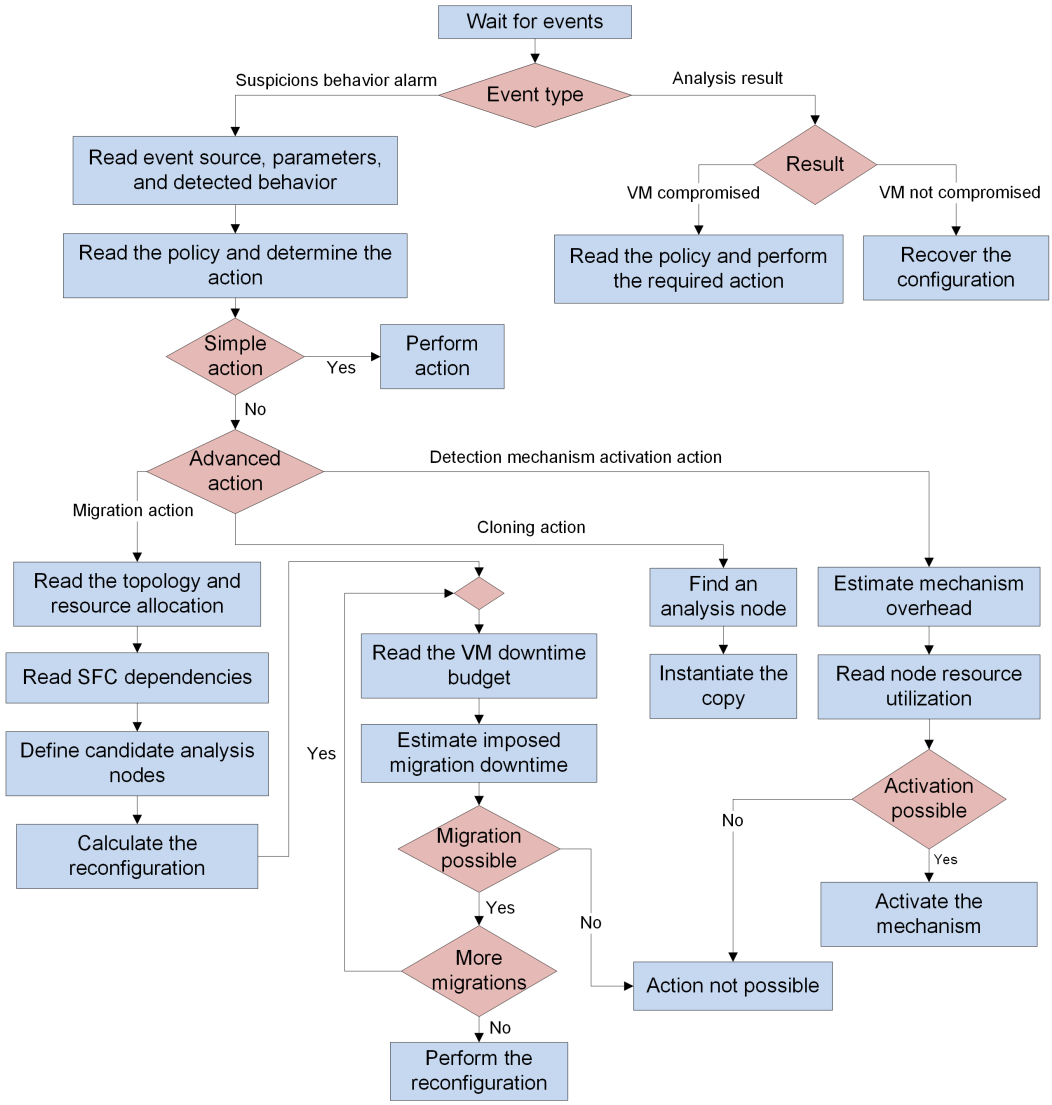

Figure 6.3: Flowchart of the decision engine algorithm

The decision engine uses the management system API to read the configuration and monitoring data and perform the required actions. If the required action is not possible for a certain reason such as the lack of resources, a simple approach is to inform the system administrator.

6.3.4 *Prototype*

The decision engine parses the user-defined policies from a policy fie using XML format. The policy defines multiple action sets per event. All action sets related to a certain event are executed (in parallel). The decision engine tries the actions in a set sequentially and only the first possible action is executed. In Listing 6.1, an exemplary policy file is presented. If VNF 53 shows a behavior pattern that refers to a side-channel attack, the decision engine first tries to migrate it to a protected host. If this action fails, the VNF is stopped. If VNF 100 shows a behavior pattern that refers to a DDOS attack, the decision

engine tries to both block the VNF and take a snapshot. If any of these action fails, the VNF is stopped.

Listing 6.1: Policy XML schema

```
<?xml version="1.0"?>
<policyFile>
    <policy>
        <VNF>53</VNF>
        <attackPattern>SideChannel</attackPattern>
        <actionSets>
            <set>
                <action>migrate</action>
                <action>stop</action>
            </set>
        </actionSets>
    </policy>
    <policy>
        <VNF>100</VNF>
        <attackPattern>DDOS</attackPattern>
        <actionSets>
            <set>
                <action>block</action>
                <action>stop</action>
            </set>
            <set>
                <action>snapshot</action>
                <action>stop</action>
            </set>
        </actionSets>
    </policy>
</policyFile>
```

The decision engine prototype is provided with a generic interface to cloud Application Programming Interfaces (APIs). A communication interface to each API type (such as Remote Procedure Call (RPC)/XML in OpenNebula) is required to parse the functions needed to monitor and configure the environment. The interface uses an environment-specific driver that defines the required API functions with their parameters and returned data structures. The interface and driver for OpenNebula have been developed.

Listing 6.2 sketches the required functions from the OpenNebula driver: the API information, VM status and operations, cluster and host status. The prototype parses the functions (using the RPC/XML interface) and replaces the missing parameters (such as $VNF - ID$) with the required values.

Listing 6.2: OpenNebula driver

```
[API_Info]
API = XMLRPC|OpenNebula
Server = http://$CloudIP$:2633/RPC2

[VM]
VM_Status_URL = one.vm.info
VM_Status_Parameter = $username$:$password$,$VMID$|int
```

```
VM_Start_URL = one.vm.action
VM_Start_Parameter = $username$:$password$,resume,$VMID$|int

VM_Stop_URL = one.vm.action
VM_Stop_Parameter = $username$:$password$,poweroff-hard,$VMID$|int

VM_Hold_URL = one.vm.action
VM_Hold_Parameter = $username$:$password$,hold,$VMID$|int

VM_Resume_URL = one.vm.action
VM_Resume_Parameter = $username$:$password$,resume,$VMID$|int

VM_Delete_URL = one.vm.action
VM_Delete_Parameter = $username$:$password$,delete,$VMID$|int

VM_Snapshot_URL = one.vm.snapshotcreate
VM_Snapshot_Parameter = $username$:$password$,$VMID$|int,$snapshotName$

VM_Migrate_URL = one.vm.migrate
VM_Migrate_Parameter = $username$:$password$,$VMID$|int,$Target$|int,True|bool,True
    |bool

[Cluster]
Cluster_Status_URL = one.cluster.info
Cluster_Status_Parameter = $username$:$password$,$CLID$|int

[Host]
Host_Status_URL = one.host.info
Host_Status_Parameter = $username$:$password$,$HOID$|int
```

6.4 SECURITY-AWARE VNE

This section includes parts of the author's publication [5]‡.

In this section, we model a basic set of security requirements of VNs. We define the topological constraints [5]‡as a new type of constraints that requires additional support by VNE. A topology constraint affects an entire subnet. For example, the VN might specify network domains that should be separated. The CDLs in this case shall be mapped through firewalls by the link mapping stage of VNE.

Another topological constraint is that virtual domains must not be split by a firewall and must be mapped onto a single physical domain. In this model, mapping the security requirements of the VLi shall check for certain properties along its physical path. The typical VNE demand/resource model has been extended to model security capabilities and demands. We model the typical security requirements (such as Trusted Hardware (TH), Network Intrusion Detection System (NIDS), and firewall) as VNE node, link, and topological demands. Furthermore, we provide a proof-of-concept implementation of this new security-aware VNE model in our VNE tool, ALEVIN, and incorporate the constraints into VNE algorithms.

6.4.1 *Scenario*

An exemplary scenario is presented in Figure 6.4 to illustrate the integration of security requirements in VNE. A cloud provider offers computing resources distributed over three data centers. Two of those data centers are protected by a firewall. One of those data centers offers two separated subnets. Domains are defined to identify the subnets that are protected by a firewall, with one domain offering NIDS. Two nodes in the SN are labeled as firewall nodes and three other nodes are labeled as providing TH.

A client wants to implement a web service that consists of a load balancer, two web servers, a database, and an authentication service. These components are interconnected and should be deployed in the cloud infrastructure. The VN represents the specified web service with three different domains: authentication and database, load balancer and web servers, and the Internet. The load balancer should connect to the Internet to receive the web requests from the service users. Intra- and inter-domain VLis are depicted.

Each of the components has its specific demands, which have to be adhered to by the cloud provider. Some of these demands reflect the security requirements of the underlying software. For example, the web servers have to be protected from the Internet by a firewall. Since a firewall cannot prevent all attacks, a NIDS should provide information about potential malicious actions. The authentication service requires TH, as it is highly security-critical. Both the authentication service and the data-base should be protected from the web servers by a firewall.

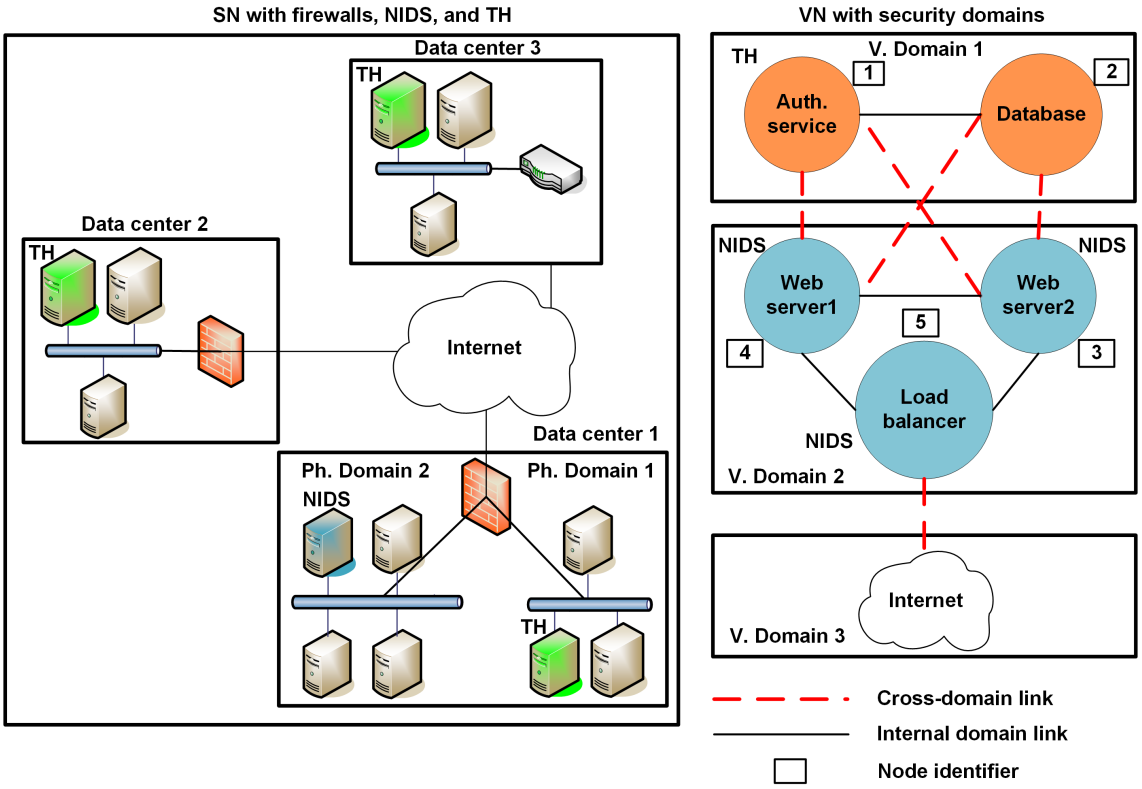

Figure 6.4: Cloud provider infrastructure and a VN for a web service [5]

6.4.2 *Implementation*

Security requirements specific to individual nodes and links can be modeled by adapting the concept of resource/demand pair. Qualitative security requirements such as "*needs protection by a NIDS*" are matched with respective security features such as "*offers protection by a NIDS*". For the presented scenario, a NIDS demand and a TH demand are implemented.

For the topological requirements, the simple resource/demand model has to be extended. The simulator has to verify the mapping path of the VLi. The path is considered valid only if the path can satisfy the security requirements of the VLi. Firewall demands are implemented through the definition of different domains. These domains are represented as identifiers that are attached to the nodes. Firewall resources are attached to the respective SNos. A check for CDLs and firewall constraints is necessary and performed by the algorithm presented in Figure 6.5.

The check is performed during the link mapping stage and forces all CDLs to go through a firewall. First, the algorithm filters the VNR to find CDLs by comparing the domain identifiers of the source and destination nodes of the VLi. Then, for each VLi, a set of possible physical paths is selected according to the link mapping method. The possible paths are then checked to assert if at least one of the nodes along the path provides a firewall service. The generic embedding algorithm enforces the following embedding constraints:

- The virtual domain is not split by a firewall and is mapped in one physical domain.
- CDLs are forced to go through a firewall.
- VNos that require a TH are mapped only to SNos that offer it.
- VNos that require a NIDS are mapped only in domains in which at least one SNo offers NIDS.

Figure 6.6 shows the results of the mapping when implementing the motivational scenario for security-aware VNE shown in Figure 6.4. To ensure readability, only mapped CDLs are represented.

The scenario is realized in ALEVIN to test the functionality of the new security-aware VNE structure and algorithm. For demonstration, a commonly known mapping algorithm is used to perform the actual embedding. Here, the vnmFlib algorithm by Lischka and Karl [194] was chosen. When the original topology does not contain firewall resources, the mapping procedure will not succeed since CDLs can only be mapped over nodes containing a firewall. However, when a firewall is added to the node that connects the first data center to the Internet, the mapping is successful.

Fig. 6.7 depicts the ALEVIN GUI with the constructed topology and the procedure of adding the property of a firewall to a SNo. For simplicity, only CDLs are shown in the VN, and only links to the firewall in the data centers and Internet links are shown in the SN. The CPU and bandwidth capacities and demands are included in the topology and allow the mapping, but are neglected from the figure for clarity.

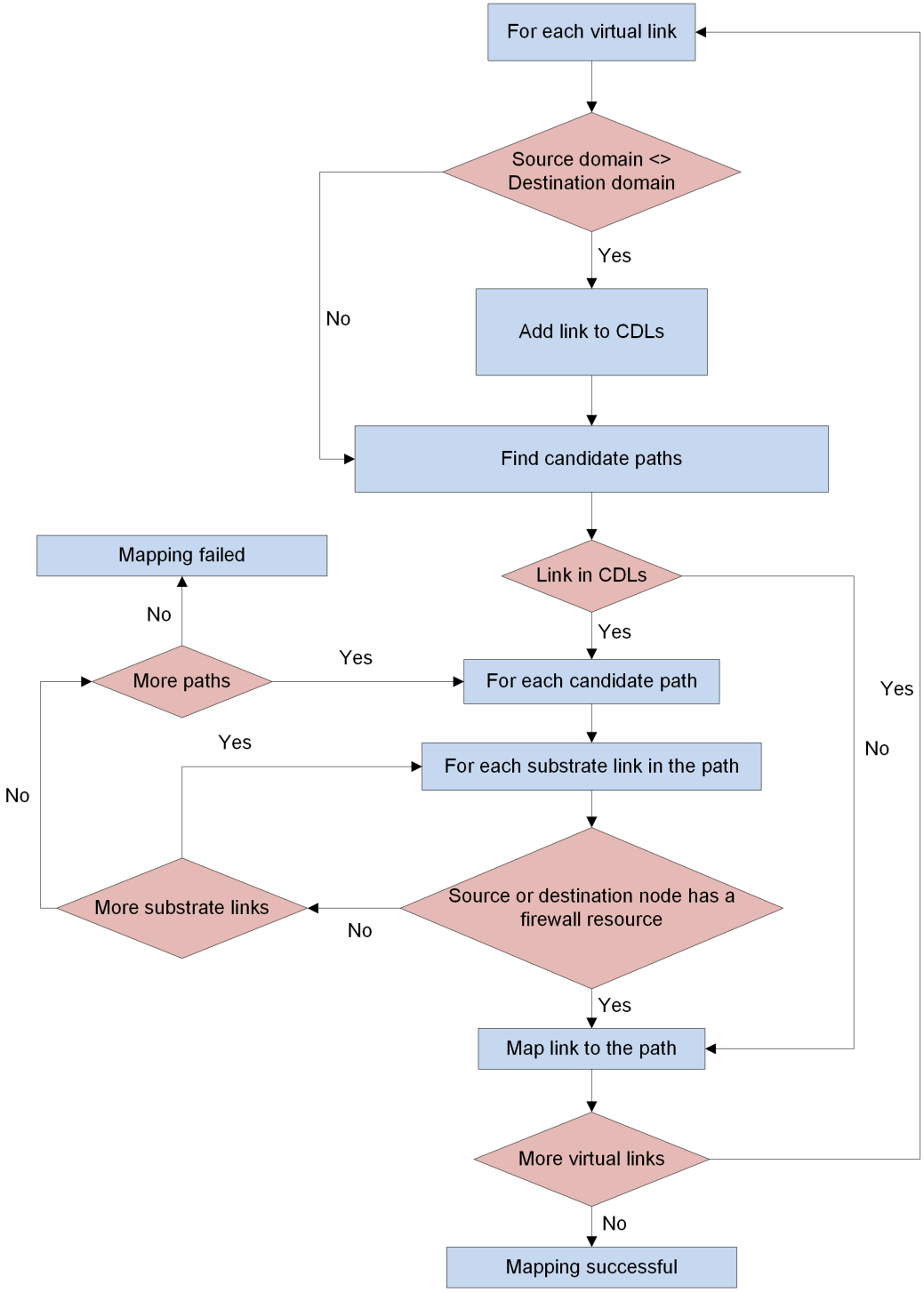

Figure 6.5: Flowchart of the embedding algorithm for CDLs [5]

6.5 CONCLUSION

In this chapter, an architecture for migration-based isolation of specious VNFs, which considers the SFC-level SLAs is presented. This defense mechanism is significant for industrial enterprises where sensitive data might be processed in shared cloud hosts. Completely integrating this solution with the EVN solution requires developing the

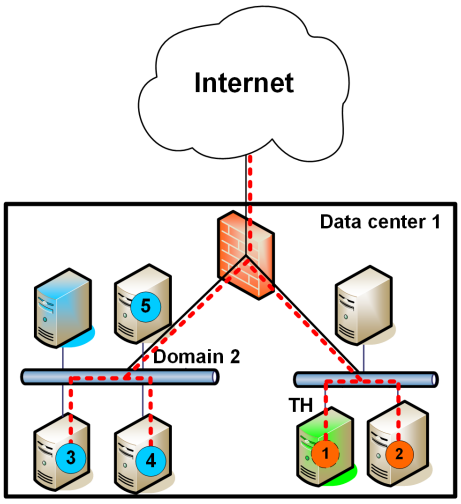

Figure 6.6: Mapping results of the motivational scenario [5]

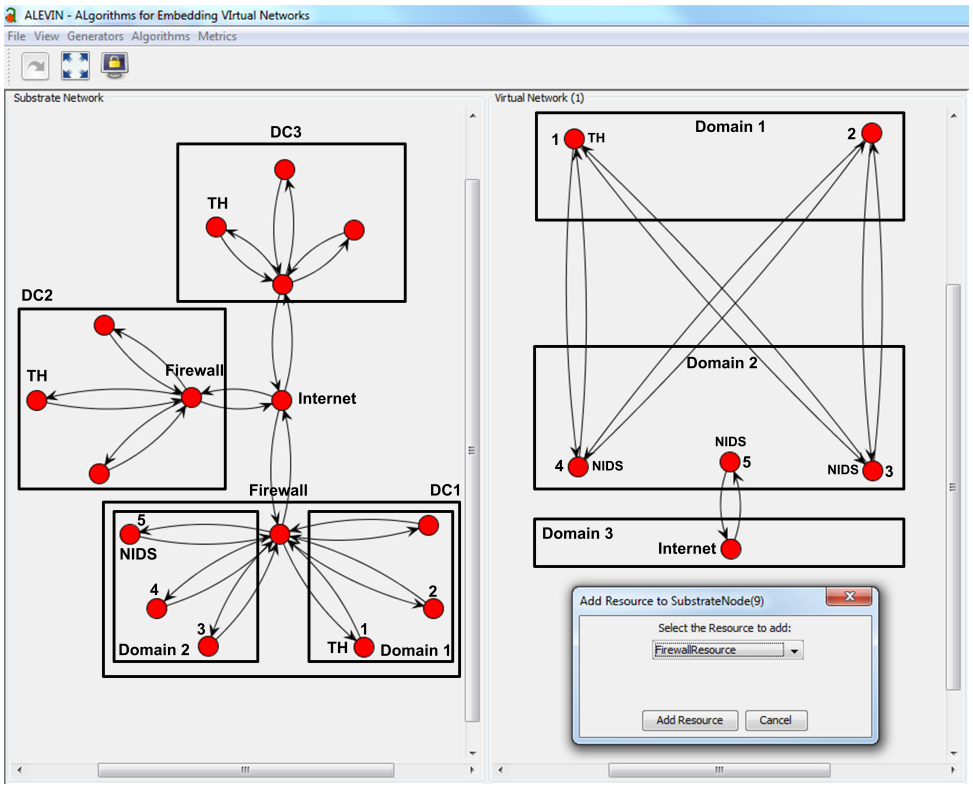

Figure 6.7: Scenario implementation in ALEVIN [5]

EVN embedding algorithm to be dynamic with on-demand reconfiguration policies as responses to failures or attacks. This aspect is a significant future work for both our EVN approach and migration-based defense approach.

Furthermore, several related QoS, privacy, and security SLA metrics and respective SFC deployment policies are proposed. Finally, a topology- and capability-aware algorithm for security-aware VNE is developed. The future work will focus on integrating the proposed SLA metrics and deployment policies completely in the prototype; considering the typical VNE evaluation scenarios; developing specific validation algorithms; integrating the security-aware VNE algorithm with the security-aware NFV presented in the combined approach; and integrating these theoretical algorithms with real security mechanisms, mainly from industrial networks.

7 CONCLUSIONS

In this chapter, we map the presented approaches in this thesis to the proposed virtualization model, as shown in Figure 7.1. This model represents the general concept of this thesis of a holistic approach of applying different network virtualization techniques to different network levels while considering the significant objectives. We map here our contributions to two primary industrial network levels (factory and enterprise), both considered virtualization dimensions (NFV and VNE), and the target objectives of topology, reliability, security, performance, and resource usage.

These objectives are applied in our work with the same order in the respective axis in the model. This order is based on best practice and our transformations in Chapter 3 where redundancy is applied before topology as a policy to reduce resource usage by only adding redundancy to the main AR VLis and not to the VLis added by the following transformations. However, this policy can be changed by the system user according to the target use case. Then, the topology transformations are applied to add the relevant devices. Then, and based on the new EVN, the required security functions are added. The performance transformations are applied at the end to add the virtual TAS if required, adapt the demands for consumable resources (bandwidth and CPU), and finally add load balancing if needed. The resource usage objective is more considered by the EVN mapping stage than during the composition stage (transformations).

Before we discuss the mapping, it shall be mentioned that a complete separation between NFV and VNE is not always possible since the SFCs are mapped using VNE algorithms. Furthermore, a complete separation between factory and enterprise levels in not possible in the main use case since the data and decisions flow among the three processing levels; factory hall, edge, and cloud computing. For these reasons, we consider approaches that are not limited to the internal factory network to be enterprise-level approaches that also process data from the factory level. In Figure 7.1, we distinguish such scenarios with the dashed-line boxes (combined with higher levels).

Furthermore, although the sub-solution of branching algorithms is used for mapping the redundant EVN links, we map it separately in this model. The sub-solution of virtual TSN is used by the combined solution (EVN) at the enterprise SFC level but discussed in Chapter 4 at the TAS VNF level, which is additionally mapped to the model. The security approaches from Chapter 6 have been used in a modified form in the combined approach and applied to industrial networks only with the EVN. We map the developed methods according to the following reasoning:

- EVN composition: applied to all objectives except resource usage, and to both network levels. However, we have a different mapping for each objective:
  - With the reliability objective, we consider transforming redundant links a VNE solution.
  - For the topology constraints, we consider adding VLis and VNos based on types of devices and locations/domain as a VNE solution.

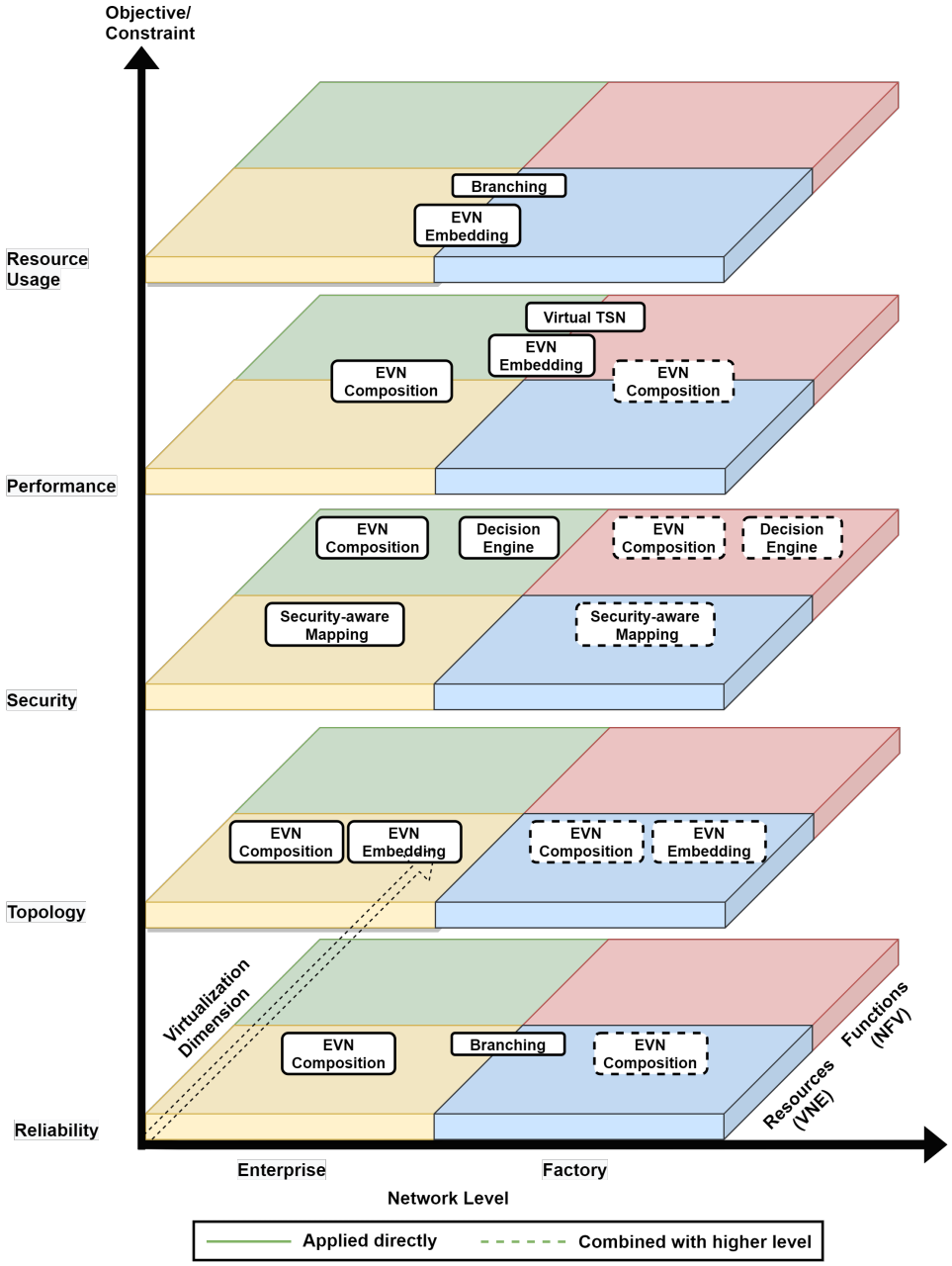

Figure 7.1: Mapping of approaches to the network virtualization model

  - With the security objective, we consider adding the required security VNFs an NFV solution.
  - With the performance objective, we consider adding the TAS VNF and adding load balancing VNF as NFV solutions. Furthermore, we consider updating the CPU and bandwidth demands in the EVN as VNE procedures.

- EVN embedding: we discuss here the traditional mapping of EVN nodes and links and chain embedding without considering the branching and security-aware mapping algorithms.

  - With the topology constraints, we consider the node mappings (devices, application VNos, VNFs) and traditional link mapping as VNE solutions that embed the composed EVN onto the SN topology.
  - With the performance objective, we consider deploying the TAS VNF on each server hosting the SFC as an NFV solution. However, the development of virtual TAS and deploying it on certain servers can also be only a factory level solution.
  - With the resource usage objective, the traditional node and link mappings and chain embedding are VNE algorithms that try to reduce the resource usage.

- Branching: The branching algorithms are VNE solutions that can be applied only inside the factory or combined with the enterprise. These algorithms are related to the reliability and resource usage objectives since the goal is either to satisfy the reliability demand with minimal resource usage or to improve it with reasonable resource usage.

- Security-aware mapping: the original security-aware mapping algorithm is a VNE solution that is mainly at the enterprise level and considers factory data.

- Decision engine: the decision engine for migration-based defense in NFVI is an NFV solution mainly at the enterprise level and considers factory data.

- Virtual TSN: The development of the virtual TAS itself is an NFV and performance solution that can be applied only inside the factory or combined with the enterprise.

In summary, we present in this thesis a base of a complete system for autonomic virtualization of an enterprise that considers all significant objectives and constraints and the existing topologies and data sources and flows. Such a system can provide the maximal benefit of the flexibility of virtualization. Furthermore, we investigate how to apply this system to a typical smart and complex enterprise represented by a use case inspired by the smart factory concept. Furthermore, we focus on TSN as a modern networking technology for deterministic latency and high reliability and integrate it into two sub-solutions. We develop several novel and efficient virtualization algorithms (heuristics) and perform in-depth evaluations. These evaluations show that: the EVN is correctly composed and mapped with low overhead; the chain embedding algorithm improves the admission with low overhead; the branching methods can improve the reliability and its demand admission with reasonable resource usage and these shall be determined based on the application and topology; virtual TAS can add high flexibility to TSN with reasonable performance for enterprise applications. Our solutions are developed to address the challenges of virtualization mentioned in Chapter 1:

- Considering multiple objectives and constraints that might be conflicting.
- Considering multiple network levels.
- Autonomic composition of environment-aware virtual networks.
- Efficient deployment algorithms.
- Applicability to complex environments.

- Integration with real, mainly modern, technologies.
- Performance degradation.
- Wider attack surface.

We propose significant future works for each approach in the respective chapter. However, a general future work is to extend the virtualization model with more objectives and constraints; compare and map it to existing domain-specific models such as RAMI 4.0; investigate its applicability to other domains such as smart grids; and use it to classify and compare network virtualization solutions and exactly identify the research gaps.

BIBLIOGRAPHY

PUBLICATIONS BY THE AUTHOR

[1] W. Mandarawi, A. Fischer, A. M. Houyou, H.-P. Huth, and H. de Meer. "Constraint-Based Virtualization of Industrial Networks." In: *Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi on his 70th Birthday*. Ed. by L. Fiondella and A. Puliafito. Springer International Publishing, 2016, pp. 567–586. ISBN: 978-3-319-30599-8. DOI: 10.1007/978-3-319-30599-8_22.

[2] W. Mandarawi, H. Chahed, and H. de Meer. "A Framework for Virtualizing Time-aware Shaper Using High Performance NFV." In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 1621–1628. DOI: 10.1109/ETFA46521.2020.9211960.

[3] W. Mandarawi, J. Rottmeier, M. Rezaeighale, and H. de Meer. "Policy-Based Composition and Embedding of Extended Virtual Networks and SFCs for IIoT." In: *Algorithms Journal; Section: Combinatorial Optimization, Graph, and Network Algorithms; Special Issue "Virtual Network Embedding"* 13.9 (2020). MDPI. ISSN: 1999-4893. DOI: 10.3390/a13090240.

[4] A. Fischer, T. Kittel, B. Kolosnjaji, T. K. Lengyel, W. Mandarawi, H. de Meer, T. Müller, M. Protsenko, H. P. Reiser, B. Taubmann, and E. Weishäupl. "CloudIDEA: A Malware Defense Architecture for Cloud Data Centers." In: *On the Move to Meaningful Internet Systems: OTM 2015 Conferences*. Springer International Publishing, 2015, pp. 594–611. ISBN: 978-3-319-26148-5.

[5] A. Fischer, R. Kühn, W. Mandarawi, and H. de Meer. "Modeling security requirements for VNE algorithms." In: *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. 2017, pp. 149–154.

[6] A. M. Houyou, A. Fischer, W. Mandarawi, H. de Meer, and H.-P. Huth. "Device and method for allocating communication resources in a system employing network slicing." In: *Google Patents*. Siemens AG. 2016. URL: https://patents.google.com/patent/WO2016150511A1/en.

[7] N. Rakotondravony, B. Taubmann, W. Mandarawi, E. Weishäupl, P. Xu, B. Kolosnjaji, M. Protsenko, H. De Meer, and H. P. Reiser. "Classifying malware attacks in IaaS cloud environments." In: *Journal of Cloud Computing, collection: Cloud Forensics and Security* 6.1 (2017). Springer, p. 26. DOI: https://doi.org/10.1186/s13677-017-0098-8.

[8] F. Ansah, S. S. P. Olaya, D. Krummacker, C. Fischer, A. Winkel, R. Guillaume, L. Wisniewski, M. Ehrlich, W. Mandarawi, H. Trsek, H. de Meer, M. Wollschlaeger, H. D. Schotten, and J. Jasperneite. "Controller of Controllers Architecture for Management of Heterogeneous Industrial Networks." In: *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. 2020, pp. 1–8. DOI: 10.1109/WFCS47810.2020.9114506.

[9] W. Mandarawi, A. Fischer, H. de Meer, and E. Weishäupl. "Qos-aware Secure Live Migration of Virtual Machines." In: *2nd International Workshop on security in highly connected IT systems (SHCIS 15)*. 2015.

[10] B. Taubmann, H. P. Reiser, T. Kittel, A. Fischer, W. Mandarawi, and H. de Meer. "CloudIDEA - Cloud Intrusion Detection, Evidence preservation and Analysis." In: *10th European Conf. on Computer Systems (EuroSys 2015)*. 2015, Poster with abstract.

REFERENCES

[11] M. T. Beck, C. Linnhoff-Popien, A. Fischer, F. Kokot, and H. de Meer. "A simulation framework for Virtual Network Embedding algorithms." In: *2014 16th International Telecommunications Network Strategy and Planning Symposium (Networks)*. 2014, pp. 1–6. DOI: 10.1109/NETWKS.2014.6959238.

[12] Y. Liu, P. Han, J. Hou, and J. Zheng. "Resource-Efficiently Survivable IoT Services Provisioning via Virtual Network Embedding in Fiber-Wireless Access Network." In: *IEEE Access* 7 (2019), pp. 65007–65018. DOI: 10.1109/ACCESS.2019.2915374.

[13] E. Sakic, V. Kulkarni, V. Theodorou, A. Matsiuk, S. Kuenzer, N. E. Petroulakis, and K. Fysarakis. "VirtuWind - An SDN- and NFV-based Architecture for Softwarized Industrial Networks." In: *CoRR* 1902.02539 abs (2019). arXiv: 1902.02539.

[14] M. Gundall, D. Reti, and H. D. Schotten. "Application of Virtualization Technologies in Novel Industrial Automation: Catalyst or Show-Stopper?" In: *Networking and Internet Architecture (cs.NI)* (2020). arXiv: 2011.07804 [cs.NI].

[15] A. Moga, T. Sivanthi, and C. Franke. "OS-Level Virtualization for Industrial Automation Systems: Are We There Yet?" In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing SAC '16*. Association for Computing Machinery, 2016, pp. 1838–1843. ISBN: 9781450337397. DOI: 10.1145/2851613.2851737.

[16] G. Bag, L. Lednicki, K. Landern, and N. Ericsson. "Experiments on Approaches of Virtualization for Industrial Internet of Things applications." In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019, pp. 1226–1229. DOI: 10.1109/ETFA.2019.8868971.

[17] J. Asenjo, J. Strohmenger, S. Nawalaniec, B. H. Hegrat, J. A. Harkulich, J. Lin Korpela, J. Rydberg Wright, R. Hessmer, J. Dyck, E. Alan Hill, and S. Conti. "Using cloud-based data for virtualization of an industrial automation environment with information overlays." In: *Google Patents*. Rockwell Automation Technologies Inc. 2013. URL: https://patents.google.com/patent/US9709978B2/en.

[18] C. Mouradian, T. Saha, J. Sahoo, M. Abu-Lebdeh, R. Glitho, M. Morrow, and P. Polakos. "Network functions virtualization architecture for gateways for virtualized wireless sensor and actuator networks." In: *IEEE Network* 30.3 (2016), pp. 72–80. ISSN: 1558-156X. DOI: 10.1109/MNET.2016.7474347.

[19] S.-H. Lee, J.-S. Kim, J.-S. Seok, and H.-W. Jin. "Virtualization of Industrial Real-Time Networks for Containerized Controllers." In: *Sensors* 19.20 (2019), p. 4405. ISSN: 1424-8220. DOI: 10.3390/s19204405.

[20] H. Huth and A. M. Houyou. "Resource-aware virtualization for industrial networks: A novel architecture combining resource management, policy control and network virtualization for networks in automation or supervisory control and data acquisition networks." In: *2013 International Conference on Data Communication Networking (DCNET)*. 2013, pp. 1–7.

[21] S. Davy, C. Fahy, L. Griffin, Z. Boudjemil, A. Berl, A. Fischer, H. Meer, and J. Strassner. "Towards a Policy-based Autonomic Virtual Network to support Differentiated Security Services." In: *International Conference on Telecommunications and Multimedia*. 2008.

[22] S. Davy, C. Fahy, Z. Boudjemil, L. Griffin, and J. Strassner. "A model based approach to autonomic management of virtual networks." In: *2009 IFIP/IEEE International Symposium on Integrated Network Management*. 2009, pp. 761–774. DOI: 10.1109/INM.2009.5188882.

[23] W. Louati, I. Houidi, and D. Zeghlache. "Autonomic Virtual Routers for the Future Internet." In: *IP Operations and Management*. Springer Berlin Heidelberg, 2009, pp. 104–115. ISBN: 978-3-642-04968-2.

[24] F. Granelli and R. Bassoli. "Autonomic Mobile Virtual Network Operators for Future Generation Networks." In: *IEEE Network* 32.5 (2018), pp. 76–84. DOI: 10.1109/MNET.2018.1700455.

[25] R. Mijumbi, J. Serrat, and J. Gorricho. "Autonomic Resource Management in Virtual Networks." In: *CoRR* abs/1503.04576 (2015). arXiv: 1503.04576.

[26] Z. Yan, J. Ge, Y. Wu, L. Li, and T. Li. "Automatic Virtual Network Embedding: A Deep Reinforcement Learning Approach With Graph Convolutional Networks." In: *IEEE Journal on Selected Areas in Communications* 38.6 (2020), pp. 1040–1057. DOI: 10.1109/JSAC.2020.2986662.

[27] A. Rkhami, T. A. Quang Pham, Y. Hadjadj-Aoul, A. Outtagarts, and G. Rubino. "On the Use of Graph Neural Networks for Virtual Network Embedding." In: *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. 2020, pp. 1–6. DOI: 10.1109/ISNCC49221.2020.9297270.

[28] I. Houidi, W. Louati, and D. Zeghlache. "Exact Multi-Objective Virtual Network Embedding in Cloud Environments." In: *The Computer Journal* 58.3 (Jan. 2015), pp. 403–415. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxu154.

[29] Z. Zhang, S. Su, Y. Lin, X. Cheng, K. Shuang, and P. Xu. "Adaptive multi-objective artificial immune system based virtual network embedding." In: *Journal of Network and Computer Applications* 53 (2015), pp. 140 –155. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.03.007.

[30] Z. Li, F. Yuan, and L. Ma. "A load balancing algorithm for solving multi-objective virtual network embedding." In: *Transactions on Emerging Telecommunications Technologies* (2020), e4066. DOI: https://doi.org/10.1002/ett.4066.

[31] L. Gong, H. Jiang, Y. Wang, and Z. Zhu. "Novel Location-Constrained Virtual Network Embedding LC-VNE Algorithms Towards Integrated Node and Link Mapping." In: *IEEE/ACM Transactions on Networking* 24.6 (2016), pp. 3648–3661. DOI: 10.1109/TNET.2016.2533625.

[32] F. Habibi, M. Dolati, A. Khonsari, and M. Ghaderi. "Accelerating Virtual Network Embedding with Graph Neural Networks." In: *2020 16th International Conference on Network and Service Management (CNSM)*. 2020, pp. 1–9. DOI: 10.23919/CNSM50824.2020.9269128.

[33] M. Chowdhury, F. Samuel, and R. Boutaba. "PolyViNE: Policy-Based Virtual Network Embedding across Multiple Domains." In: *Proceedings of the Second ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures VISA '10*. Association for Computing Machinery, 2010, pp. 49–56. ISBN: 9781450301992. DOI: 10.1145/1851399.1851408.

[34] D. Dietrich, A. Rizk, and P. Papadimitriou. "Multi-domain virtual network embedding with limited information disclosure." In: *2013 IFIP Networking Conference*. 2013, pp. 1–9.

[35] Y. Ni, G. Huang, S. Wu, C. Li, P. Zhang, and H. Yao. "A PSO based multi-domain virtual network embedding approach." In: *China Communications* 16.4 (2019), pp. 105–119.

[36] D. Andreoletti, T. Velichkova, G. Verticale, M. Tornatore, and S. Giordano. "A Privacy-Preserving Reinforcement Learning Algorithm for Multi-Domain Virtual Network Embedding." In: *IEEE Transactions on Network and Service Management* 17.4 (2020), pp. 2291–2304. DOI: 10.1109/TNSM.2020.3022278.

[37] P. Zhang, C. Wang, Z. Qin, and H. Cao. "A multidomain virtual network embedding algorithm based on multiobjective optimization for Internet of Drones architecture in Industry 4.0." In: *Software: Practice and Experience* (2020). DOI: https://doi.org/10.1002/spe.2815.

[38] M. Yu, Y. Yi, J. Rexford, and M. Chiang. "Rethinking virtual network embedding: substrate support for path splitting and migration." In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 17–29.

[39] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann. "Vnr algorithm: A greedy approach for virtual networks reconfigurations." In: *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011*. IEEE. 2011, pp. 1–6.

[40] J. G. Herrera and J. F. Botero. "Resource Allocation in NFV: A Comprehensive Survey." In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 518–532. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2598420.

[41] S. Mehraghdam, M. Keller, and H. Karl. "Specifying and placing chains of virtual network functions." In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. 2014, pp. 7–13. DOI: 10.1109/CloudNet.2014.6968961.

[42] A. F. Ocampo, J. Gil-Herrera, P. H. Isolani, M. C. Neves, J. F. Botero, S. Latré, L. Zambenedetti, M. P. Barcellos, and L. P. Gaspary. "Optimal Service Function Chain Composition in Network Functions Virtualization." In: *Security of Networks and Services in an All-Connected World: 11th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2017*. Springer International Publishing, 2017, pp. 62–76. ISBN: 978-3-319-60774-0. DOI: 10.1007/978-3-319-60774-0_5.

[43] J. Gil-Herrera and J. F. Botero. "A scalable metaheuristic for service function chain composition." In: *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*. 2017, pp. 1–6. DOI: 10.1109/LATINCOM.2017.8240194.

[44] D. Zheng, C. Peng, X. Liao, L. Tian, G. Luo, and X. Cao. "Towards Latency Optimization in Hybrid Service Function Chain Composition and Embedding." In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2020, pp. 1539–1548. DOI: 10.1109/INFOCOM41043.2020.9155529.

[45] M. Wang, B. Cheng, B. Li, and J. Chen. "Service Function Chain Composition and Mapping in NFV-Enabled Networks." In: *2019 IEEE World Congress on Services (SERVICES)*. Vol. 2642-939X. 2019, pp. 331–334. DOI: 10.1109/SERVICES.2019.00092.

[46] M. T. Beck and J. F. Botero. "Coordinated Allocation of Service Function Chains." In: *2015 IEEE Global Communications Conference (GLOBECOM)*. 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2015.7417401.

[47] A. Hirwe and K. Kataoka. "LightChain: A lightweight optimisation of VNF placement for service chaining in NFV." In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. 2016, pp. 33–37. DOI: 10.1109/NETSOFT.2016.7502438.

[48] Z. Wang, J. Zhang, T. Huang, and Y. Liu. "Service Function Chain Composition, Placement, and Assignment in Data Centers." In: *IEEE Transactions on Network and Service Management* 16.4 (2019), pp. 1638–1650.

[49] D. Li, P. Hong, K. Xue, and J. Pei. "Virtual Network Function Placement Considering Resource Optimization and SFC Requests in Cloud Datacenter." In: *IEEE Transactions on Parallel and Distributed Systems* (2018), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2802518.

[50] Y. Wang, Z. Li, G. Xie, and K. Salamatian. "Enabling automatic composition and verification of service function chain." In: *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. 2017, pp. 1–5.

[51] H. Li, L. Wang, X. Wen, Z. Lu, and L. Ma. "Constructing Service Function Chain Test Database: An Optimal Modeling Approach for Coordinated Resource Allocation." In: *IEEE Access* 6 (2018), pp. 17595–17605.

[52] M. Wang, B. Cheng, S. Zhao, B. Li, W. Feng, and J. Chen. "Availability-Aware Service Chain Composition and Mapping in NFV-Enabled Networks." In: *2019 IEEE International Conference on Web Services (ICWS)*. 2019, pp. 107–115.

[53] M. T. Beck, J. F. Botero, and K. Samelin. "Resilient allocation of service Function chains." In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2016, pp. 128–133. DOI: 10.1109/NFV-SDN.2016.7919487.

[54] N. Torkzaban and J. S. Baras. "Trust-Aware Service Function Chain Embedding: A Path-Based Approach." In: *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2020, pp. 31–36. DOI: 10.1109/NFV-SDN50289.2020.9289885.

[55] Y. Liu, H. Zhang, D. Chang, and H. Hu. "GDM: A General Distributed Method for Cross-Domain Service Function Chain Embedding." In: *IEEE Transactions on Network and Service Management* 17.3 (2020), pp. 1446–1459. DOI: 10.1109/TNSM.2020.2993364.

[56] S. Lange, A. Grigorjew, T. Zinner, P. Tran-Gia, and M. Jarschel. "A Multi-objective Heuristic for the Optimization of Virtual Network Function Chain Placement." In: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 1. 2017, pp. 152–160. DOI: 10.23919/ITC.2017.8064351.

[57] *Intel Data plane development kit*. URL: https://doc.dpdk.org. (Online; accessed: 20.04.2020).

[58] C. Sun, J. Bi, Z. Zheng, and H. Hu. "HYPER: A hybrid high-performance framework for network function virtualization." In: *IEEE Journal on Selected Areas in Communications* 35.11 (2017), pp. 2490–2500.

[59] P. Naik, A. Kanase, T. Patel, and M. Vutukuru. "libVNF: building virtual network functions made easy." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 212–224.

[60] Y. Nakajima, H. Masutani, and H. Takahashi. "High-Performance vNIC Framework for Hypervisor-Based NFV with Userspace vSwitch." In: *2015 Fourth European Workshop on Software Defined Networks*. 2015, pp. 43–48. DOI: 10.1109/EWSDN.2015.59.

[61] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal. "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration." In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. 2015, pp. 74–78.

[62] Z. Li. "HPSRouter: A high performance software router based on DPDK." In: *2018 20th International Conference on Advanced Communication Technology (ICACT)*. IEEE. 2018, pp. 503–506.

[63] M. Yurchenko, P. Cody, A. Coplan, R. Kennedy, T. Wood, and K. K. Ramakrishnan. "OpenNetVM: A Platform for High Performance NFV Service Chains." In: *Proceedings of the Symposium on SDN Research SOSR '18*. Association for Computing Machinery, 2018. ISBN: 9781450356640. DOI: 10.1145/3185467.3190786.

[64] K. Yasukata, F. Huici, V. Maffione, G. Lettieri, and M. Honda. "HyperNF: Building a High Performance, High Utilization and Fair NFV Platform." In: *Proceedings of the 2017 Symposium on Cloud Computing SoCC '17*. Association for Computing Machinery, 2017, pp. 157–169. ISBN: 9781450350280. DOI: 10.1145/3127479.3127489.

[65] C. Sun, J. Bi, Z. Zheng, and H. Hu. "SLA-NFV: An SLA-Aware High Performance Framework for Network Function Virtualization." In: *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 581–582. ISBN: 9781450341936. DOI: 10.1145/2934872.2959058.

[66] Z. Zheng, J. Bi, C. Sun, H. Yu, H. Hu, Z. Meng, S. Wang, K. Gao, and J. Wu. "GEN: A GPU-Accelerated Elastic Framework for NFV." In: *Proceedings of the 2nd Asia-Pacific Workshop on Networking APNet '18*. Association for Computing Machinery, 2018, pp. 57–64. ISBN: 9781450363952. DOI: 10.1145/3232565.3234510.

[67] J. Jiang, Y. Li, S. H. Hong, A. Xu, and K. Wang. "A time-sensitive networking (TSN) simulation model based on OMNET++." In: *2018 IEEE International Conference on Mechatronics and Automation (ICMA)*. IEEE. 2018, pp. 643–648.

[68] *OMNeT++, discrete event simulator, official website*. URL: https://omnetpp.org/. (Online; accessed: 20.04.2020).

[69] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel. "NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++." In: *2019 International Conference on Networked Systems (NetSys)*. IEEE. 2019, pp. 1–8.

[70] P. Heise, F. Geyer, and R. Obermaisser. "TSimNet: An industrial time sensitive networking simulation framework based on OMNeT++." In: *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE. 2016, pp. 1–5.

[71] *INET Framework official website*. URL: http://https://inet.omnetpp.org//. (Online; accessed: 20.04.2020).

[72] M. Pahlevan. "Time sensitive networking for virtualized integrated real-time systems." In: *Dissertation, Doctor of Engineering*. University of Siegen. 2019.

[73] J. Lee and S. Park. "Time-Sensitive Network (TSN) Experiment in Sensor-Based Integrated Environment for Autonomous Driving." In: *Sensors* 19.5 (2019), p. 1111.

[74] M. Pahlevan and R. Obermaisser. "Redundancy management for safety-critical applications with time sensitive networking." In: *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*. IEEE. 2018, pp. 1–7.

[75] M. Pahlevan and R. Obermaisser. "Evaluation of time-triggered traffic in time-sensitive networks using the opnet simulation framework." In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 283–287.

[76] S. Brooks and E. Uludag. "Time-Sensitive Networking: From Theory to Implementation in Industrial Automation." In: *White Paper*. Intel. 2019.

[77] *Broadcom Introduces First Fully Compliant TSN Ethernet Switch*. 2017. URL: https://www.broadcom.com/company/news/product-releases/12041. (Online; accessed: 20.04.2020).

[78] "Time-Sensitive Networking: A Technical Introduction." In: *White Paper*. Cisco. 2017.

[79] M. Gundall, C. Huber, P. Rost, R. Halfmann, and H. D. Schotten. "Integration of 5G with TSN as Prerequisite for a Highly Flexible Future Industrial Automation: Time Synchronization based on IEEE 802.1AS." In: *46th Annual Conference of the IEEE Industrial Electronics Society (IECON-2020)*. IEEE, Oct. 2020.

[80] "5G-TSN integration meets networking requirements for industrial automation." In: *ERICSSON Technology Review*. 2019.

[81] P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner. "Enabling Fog Computing for Industrial Automation Through Time-Sensitive Networking (TSN)." In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 55–61. DOI: 10.1109/MCOMSTD.2018.1700057.

[82] Y. Li, J. Jiang, C. Lee, and S. H. Hong. "Practical Implementation of an OPC UA TSN Communication Architecture for a Manufacturing System." In: *IEEE Access* 8 (2020), pp. 200100–200111. DOI: 10.1109/ACCESS.2020.3035548.

[83] S. B. H. Said, Q. H. Truong, and M. Boc. "SDN-Based Configuration Solution for IEEE 802.1 Time Sensitive Networking (TSN)." In: *ACM SIGBED Rev.* 16.1 (Feb. 2019), pp. 27–32. DOI: 10.1145/3314206.3314210.

[84] M. Boehm, J. Ohms, M. Kumar, O. Gebauer, and D. Wermser. "Time-Sensitive Software-Defined Networking: A Unified Control- Plane for TSN and SDN." In: *Mobile Communication - Technologies and Applications; 24. ITG-Symposium*. 2019, pp. 1–6.

[85] H. Fang and R. Obermaisser. "Virtual Switch for Integrated Real-Time Systems based on SDN." In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. 2019, pp. 344–351. DOI: 10.1109/IOTSMS48152.2019.8939207.

[86] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner. "Scheduling real-time communication in IEEE 802.1 Qbv time sensitive networks." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 183–192.

[87] M. H. Farzaneh, S. Kugele, and A. Knoll. "A graphical modeling tool supporting automated schedule synthesis for time-sensitive networking." In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2017, pp. 1–8.

[88] V. Gavriluţ and P. Pop. "Scheduling in time sensitive networks (TSN) for mixed-criticality industrial applications." In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. IEEE. 2018, pp. 1–4.

[89] M. Pahlevan, N. Tabassam, and R. Obermaisser. "Heuristic list scheduler for time triggered traffic in time sensitive networks." In: *ACM Sigbed Review* 16.1 (2019), pp. 15–20.

[90] F. Dürr and N. G. Nayak. "No-wait packet scheduling for IEEE time-sensitive networks (TSN)." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 203–212.

[91] M. Barzegaran, B. Zarrin, and P. Pop. "Quality-Of-Control-Aware Scheduling of Communication in TSN-Based Fog Computing Platforms Using Constraint Programming." In: *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Vol. 80. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:9. ISBN: 978-3-95977-144-3. DOI: 10.4230/OASIcs.Fog-IoT.2020.3.

[92] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, and F. Drr. "Scaling TSN Scheduling for Factory Automation Networks." In: *2020 16th IEEE International Conference on Factory Communication Systems (WFCS)*. 2020, pp. 1–8. DOI: 10.1109/WFCS47810.2020.9114415.

[93] A. Arestova, K.-S. J. Hielscher, and R. German. "Design of a Hybrid Genetic Algorithm for Time-Sensitive Networking." In: *Measurement, Modelling and Evaluation of Computing Systems*. Springer International Publishing, 2020, pp. 99–117. ISBN: 978-3-030-43024-5.

[94] R. Serna Oliver, S. S. Craciunas, and W. Steiner. "IEEE 802.1Qbv Gate Control List Synthesis Using Array Theory Encoding." In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2018, pp. 13–24. DOI: 10.1109/RTAS.2018.00008.

[95] A. Jarray, Y. Song, and A. Karmouch. "Resilient virtual network embedding." In: *2013 IEEE International Conference on Communications (ICC)*. 2013, pp. 3461–3465. DOI: 10.1109/ICC.2013.6655085.

[96] Y. Chen, J. Li, T. Wo, C. Hu, and W. Liu. "Resilient Virtual Network Service Provision in Network Virtualization Environments." In: *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. 2010, pp. 51–58. DOI: 10.1109/ICPADS.2010.26.

[97] H. Jiang, L. Gong, and Z. W. Zuqing. "Efficient joint approaches for location-constrained survivable virtual network embedding." In: *2014 IEEE Global Communications Conference*. 2014, pp. 1810–1815. DOI: 10.1109/GLOCOM.2014.7037071.

[98] R. R. Oliveira, D. S. Marcon, L. R. Bays, M. C. Neves, L. S. Buriol, L. P. Gaspary, and M. P. Barcellos. "No more backups: Toward efficient embedding of survivable virtual networks." In: *2013 IEEE International Conference on Communications (ICC)*. 2013, pp. 2128–2132. DOI: 10.1109/ICC.2013.6654841.

[99] R. Chai, D. Xie, L. Luo, and Q. Chen. "Multi-Objective Optimization-Based Virtual Network Embedding Algorithm for Software-Defined Networking." In: *IEEE Transactions on Network and Service Management* 17.1 (2020), pp. 532–546. DOI: 10.1109/TNSM.2019.2953297.

[100] M. R. Rahman and R. Boutaba. "SVNE: Survivable Virtual Network Embedding Algorithms for Network Virtualization." In: *IEEE Transactions on Network and Service Management* 10.2 (2013), pp. 105–118. DOI: 10.1109/TNSM.2013.013013.110202.

[101] B. Guo, C. Qiao, J. Wang, H. Yu, Y. Zuo, J. Li, Z. Chen, and Y. He. "Survivable Virtual Network Design and Embedding to Survive a Facility Node Failure." In: *Journal of Lightwave Technology* 32.3 (2014), pp. 483–493. DOI: 10.1109/JLT.2013.2293193.

[102] X. Liu, X. Wang, D. Yuan, and M. Chen. "A scheme of survival virtual network embedding based on robust topology design." In: *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*. 2017, pp. 993–998. DOI: 10.1109/ICCSN.2017.8230259.

[103] D. Ergenc, J. Rak, and M. Fischer. "Service-Based Resilience for Embedded IoT Networks." In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2020, pp. 540–551. DOI: 10.1109/DSN48063.2020.00066.

[104] Y. Chen, S. Ayoubi, and C. Assi. "CORNER: COst-Efficient and Reliability Aware Virtual NEtwork Redesign and Embedding." In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. 2014, pp. 356–361. DOI: 10.1109/CloudNet.2014.6969021.

[105] D. Wu, Z. Liu, Z. Yang, P. Zhang, R. Wang, and X. Ma. "Survivability-Enhanced Virtual Network Embedding Strategy in Virtualized Wireless Sensor Networks." In: *Sensors* 21.1 (2021). ISSN: 1424-8220. DOI: 10.3390/s21010218.

[106] P. Zhang, S. Wu, M. Wang, H. Yao, and Y. Liu. "Topology based reliable virtual network embedding from a QoE perspective." In: *China Communications* 15.10 (2018), pp. 38–50. DOI: 10.1109/CC.2018.8485467.

[107] M. R. Rahman, I. Aib, and R. Boutaba. "Survivable virtual network embedding." In: *International Conference on Research in Networking*. Springer. 2010, pp. 40–52.

[108] R. L. Gomes, L. F. Bittencourt, and E. R. Madeira. "A bandwidth-feasibility algorithm for reliable virtual network allocation." In: *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. IEEE. 2014, pp. 504–511.

[109] H. Yu, V. Anand, C. Qiao, and H. Di. "Migration based protection for virtual infrastructure survivability for link failure." In: *Optical Fiber Communication Conference*. Optical Society of America. 2011, OTuR2.

[110] L. Xing, G. Levitin, and Y. Xiang. "Defending N-version Programming Service Components against Co-resident Attacks in IoT Cloud Systems." In: *IEEE Transactions on Services Computing* (2019), pp. 1–1. DOI: 10.1109/TSC.2019.2904958.

[111] A. Dolgikh, Z. Birnbaum, Y. Chen, and V. Skormin. "Behavioral Modeling for Suspicious Process Detection in Cloud Computing Environments." In: *IEEE 14th Int. Conf. on Mobile Data Management (MDM)*. Vol. 2. 2013, pp. 177–181. DOI: 10.1109/MDM.2013.90.

[112] A. Marnerides, M. Watson, N. Shirazi, A. Mauthe, and D. Hutchison. "Malware analysis in cloud computing: Network and system characteristics." In: *Globecom Workshops (GC Wkshps), 2013 IEEE*. 2013, pp. 482–487. DOI: 10.1109/GLOCOMW.2013.6825034.

[113] Y. Zhang, A. Juels, A. Oprea, and M. Reiter. "HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis." In: *IEEE Sympl. on Security and Privacy*. 2011, pp. 313–328. DOI: 10.1109/SP.2011.31.

[114] W. Hou, Z. Ning, L. Guo, Z. Chen, and M. S. Obaidat. "Novel Framework of Risk-Aware Virtual Network Embedding in Optical Data Center Networks." In: *IEEE Systems Journal* 12.3 (2018), pp. 2473–2482. DOI: 10.1109/JSYST.2017.2673828.

[115] M. A. Elshabka, H. A. Hassan, W. M. Sheta, and H. M. Harb. "Security-aware dynamic VM consolidation." In: *Egyptian Informatics Journal* (2020). ISSN: 1110-8665. DOI: https://doi.org/10.1016/j.eij.2020.10.002.

[116] A. Bardas, Sundaramurthy, S.C., X. Ou, and S. DeLoach. "MTD CBITS: Moving Target Defense for Cloud-Based IT Systems." In: *Computer Security – ESORICS 2017. Lecture Notes in Computer Science*. Vol. 10492. Springer, Cham, 2017. DOI: 10.1007/978-3-319-66402-6_11.

[117] Y. Han, T. Alpcan, J. Chan, C. Leckie, and B. I. P. Rubinstein. "A Game Theoretical Approach to Defend Against Co-Resident Attacks in Cloud Computing: Preventing Co-Residence Using Semi-Supervised Learning." In: *IEEE Transactions on Information Forensics and Security* 11.3 (2016), pp. 556–570. DOI: 10.1109/TIFS.2015.2505680.

[118] Y. Han, T. Alpcan, J. Chan, and C. Leckie. "Security Games for Virtual Machine Allocation in Cloud Computing." In: *Decision and Game Theory for Security*. Springer International Publishing, 2013, pp. 99–118. ISBN: 978-3-319-02786-9.

[119] F. Miao, L. Wang, and Z. Wu. "A VM Placement Based Approach to Proactively Mitigate Co-Resident Attacks in Cloud." In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. 2018, pp. 00285–00291. DOI: 10.1109/ISCC.2018.8538543.

[120] A. Agarwal and T. N. Binh Duong. "Co- Location Resistant Virtual Machine Placement in Cloud Data Centers." In: *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 2018, pp. 61–68. DOI: 10.1109/PADSW.2018.8644849.

[121] S. Feizollahibarough and M. Ashtiani. "A security-aware virtual machine placement in the cloud using hesitant fuzzy decision-making processes." In: *J Supercomput* (2020). DOI: 10.1007/s11227-020-03496-4.

[122] M. Azab and M. Eltoweissy. "MIGRATE: Towards a Lightweight Moving-Target Defense Against Cloud Side-Channels." In: *2016 IEEE Security and Privacy Workshops (SPW)*. 2016, pp. 96–103. DOI: 10.1109/SPW.2016.28.

[123] Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang. "Incentive Compatible Moving Target Defense against VM-Colocation Attacks in Clouds." In: *Information Security and Privacy Research*. Springer Berlin Heidelberg, 2012, pp. 388–399. ISBN: 978-3-642-30436-1.

[124] A. H. Anwar, G. Atia, and M. Guirguis. "It's Time to Migrate! A Game-Theoretic Framework for Protecting a Multi-Tenant Cloud against Collocation Attacks." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 725–731. DOI: 10.1109/CLOUD.2018.00099.

[125] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel. "Malicious co-residency on the cloud: Attacks and defense." In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8056951.

[126] Y. Liu, X. Ruan, S. Cai, R. Li, and H. He. "An optimized VM Allocation Strategy to Make a Secure and Energy-Efficient Cloud Against Co-residence Attack." In: *2018 International Conference on Computing, Networking and Communications (ICNC)*. 2018, pp. 349–353. DOI: 10.1109/ICCNC.2018.8390415.

[127] H. S. Bedi and S. Shiva. "Securing Cloud Infrastructure against Co-Resident DoS Attacks Using Game Theoretic Defense Mechanisms." In: *Proceedings of the International Conference on Advances in Computing, Communications and Informatics ICACCI '12*. Association for Computing Machinery, 2012, pp. 463–469. ISBN: 9781450311960. DOI: 10.1145/2345396.2345473.

[128] D. Deyannis, E. Papadogiannaki, G. Kalivianakis, G. Vasiliadis, and S. Ioannidis. "TrustAV: Practical and Privacy Preserving Malware Analysis in the Cloud." In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. Association for Computing Machinery, 2020, pp. 39–48. ISBN: 9781450371070. DOI: 10.1145/3374664.3375748.

[129] V. Casola, A. De Benedictis, M. Rak, and U. Villano. "A Security SLA-Driven Moving Target Defense Framework to Secure Cloud Applications." In: *Proceedings of the 5th ACM Workshop on Moving Target Defense*. Association for Computing Machinery, 2018, pp. 48–56. ISBN: 9781450360036. DOI: 10.1145/3268966.3268975.

[130] N. Kaaniche, M. Mohamed, M. Laurent, and H. Ludwig. "Security SLA Based Monitoring in Clouds." In: *2017 IEEE International Conference on Edge Computing (EDGE)*. 2017, pp. 90–97. DOI: 10.1109/IEEE.EDGE.2017.20.

[131] A. De Benedictis, V. Casola, M. Rak, and U. Villano. "Cloud Security: From Per-Provider to Per-Service Security SLAs." In: *2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)*. 2016, pp. 469–474. DOI: 10.1109/INCoS.2016.61.

[132] S. Zhou, L. Wu, and C. Jin. "A privacy-based SLA violation detection model for the security of cloud computing." In: *China Communications* 14.9 (2017), pp. 155–165. DOI: 10.1109/CC.2017.8068773.

[133] C. Basile, A. Lioy, C. Pitscheider, F. Valenza, and M. Vallini. "A novel approach for integrating security policy enforcement with dynamic network virtualization." In: *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. 2015, pp. 1–5. DOI: 10.1109/NETSOFT.2015.7116152.

[134] K. W. Ullah and A. S. Ahmed. "Demo Paper: Automatic Provisioning, Deploy and Monitoring of Virtual Machines Based on Security Service Level Agreement in the Cloud." In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2014, pp. 536–537. DOI: 10.1109/CCGrid.2014.39.

[135] L. R. Bays, R. R. Oliveira, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary. "Security-aware optimal resource allocation for virtual network embedding." In: *2012 8th international conference on network and service management (cnsm) and 2012 workshop on systems virtualiztion management (svm)*. 2012, pp. 378–384.

[136] L. R. Bays, R. R. Oliveira, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary. "A heuristic-based algorithm for privacy-oriented virtual network embedding." In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. 2014, pp. 1–8. DOI: 10.1109/NOMS.2014.6838360.

[137] P. Zhang, C. Wang, C. Jiang, and A. Benslimane. "Security-Aware Virtual Network Embedding Algorithm based on Reinforcement Learning." In: *IEEE Transactions on Network Science and Engineering* (2020). DOI: 10.1109/TNSE.2020.2995863.

[138] Y. Wang, P. Chau, and F. Chen. "Towards a secured network virtualization." In: *Computer Networks* 104 (2016), pp. 55 –65. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2016.04.023.

[139] S. Liu, Z. Cai, H. Xu, and M. Xu. "Security-aware virtual network embedding." In: *2014 IEEE International Conference on Communications (ICC)*. IEEE. 2014, pp. 834–840.

[140] C. Beşiktaş, D. Gözüpek, A. Ulaş, and E. Lokman. "Secure virtual network embedding with flexible bandwidth-based revenue maximization." In: *Computer Networks* 121 (2017), pp. 89 –99. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.04.020.

[141] S. Liu, Z. Cai, H. Xu, and M. Xu. "Towards security-aware virtual network embedding." In: *Computer Networks* 91 (2015), pp. 151 –163. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2015.08.014.

[142] Y. Wang, P. Chau, and F. Chen. "A Framework for Security-Aware Virtual Network Embedding." In: *2015 24th International Conference on Computer Communication and Networks (ICCCN)*. 2015, pp. 1–7. DOI: 10.1109/ICCCN.2015.7288361.

[143] *IBM - 5 top industrial IoT use cases*. 2017. URL: https://www.ibm.com/blogs/internet-of-things/top-5-industrial-iot-use-cases/. (Online; accessed: 20.04.2020).

[144] M. Aazam, S. Zeadally, and K. A. Harras. "Deploying fog computing in industrial internet of things and industry 4.0." In: *IEEE Transactions on Industrial Informatics* 14.10 (2018), pp. 4674–4682.

[145] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Books, 1997. ISBN: 9789812384720. URL: https://books.google.de/books?id=P3r82gfRf8MC.

[146] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou. "P4SC: Towards High-Performance Service Function Chain Implementation on the P4-Capable Device." In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2019, pp. 1–9.

[147] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen, 4. Auflage*. Book - Spektrum Akademischer Verlag, 2002. ISBN: 3-8274-1029-0.

[148] D. Eppstein. "Finding the k shortest paths." In: *SIAM Journal on computing* 28.2 (1998), pp. 652–673.

[149] B. M. Waxman. "Routing of multipoint connections." In: *IEEE Journal on Selected Areas in Communications* 6.9 (1988), pp. 1617–1622. ISSN: 0733-8716. DOI: 10.1109/49.12889.

[150] R. Albert and A.-L. Barabási. "Statistical mechanics of complex networks." In: *Rev. Mod. Phys.* 74 (1 2002), pp. 47–97. DOI: 10.1103/RevModPhys.74.47.

[151] "IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks." In: *IEEE Std 802.1Q-2018* (2018).

[152] "Network functions virtualization (nfv) infrastructure overview." In: *ETSI GS NFV-INF*. ETSI. 2015.

[153] "Network Functions Virtualisation." In: *White Paper*. ETSI. 2014.

[154] Đ. Vladislavić, D. Huljenić, and J. Ožegović. "Enhancing VNF's performance using DPDK driven OVS user-space forwarding." In: *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE. 2017, pp. 1–5.

[155] S. Shanmugalingam, A. Ksentini, and P. Bertin. "DPDK Open vSwitch performance validation with mirroring feature." In: *2016 23rd International Conference on Telecommunications (ICT)*. IEEE. 2016, pp. 1–6.

[156] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems." In: *IEEE Std 1588-2019* (2019).

[157] M. Hosamo. "A study of the source traffic generator using poisson distribution for ABR service." In: *Modelling and Simulation in Engineering* 2012 (2012).

[158] D.-C. Juan, L. Li, H.-K. Peng, D. Marculescu, and C. Faloutsos. "Beyond poisson: Modeling inter-arrival time of requests in a datacenter." In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2014, pp. 198–209.

[159] *Open vSwitch official website*. URL: http://www.openvswitch.org/. (Online; accessed: 20.04.2020).

[160] *Red Hat Solution for NFV*. URL: https://access.redhat.com/. (Online; accessed: 20.04.2020).

[161] *OpenStack - Choosing a hypervisor*. URL: https://docs.openstack.org/arch-design/design-compute/design-compute-hypervisor.html. (Online; accessed: 20.04.2020).

[162] "IEEE Standard for Local and metropolitan area networks - Frame Replication and Elimination for Reliability." In: *IEEE Std 802.1CB-2017* (2017). DOI: 10.1109/IEEESTD.2017.8091139.

[163] "IEEE Standard for Local and metropolitan area networks - Frame Replication and Elimination for Reliability - Amendment: Extended Stream identification functions - Draft." In: *IEEE P802.1CBdb* (2019).

[164] *Project German BMBF FIND - Future Industrial Network Architecture*. http://www.future-industrial-internet.de. 2017-2020.

[165] M. Čepin. "Reliability Block Diagram." In: *Assessment of Power System Reliability: Methods and Applications*. Springer, 2011, pp. 119–123. ISBN: 978-0-85729-688-7. DOI: 10.1007/978-0-85729-688-7_9.

[166] P. Heise. "Real-time guarantees, dependability and self-configuration in future avionic networks." In: *Dissertation, Doctor of Engineering*. University of Siegen. 2018.

[167] M. S. Inci, B. Gulmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar. "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud." In: *IACR Cryptology ePrint Archive* (2015), p. 898.

[168] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *IEEE Symposium on Security and Privacy (SP)*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[169] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds." In: *Proceedings of the 16th ACM Conference on Computer and Communications Security CCS '09*. ACM, 2009, pp. 199–212. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687.

[170] D. Song, D. Wagner, and X. Tian. "Timing analysis of keystrokes and SSH timing attacks." In: *10th USENIX Security Symposium*. 2001, pp. 337–352.

[171] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar. "Fine Grain Cross-VM Attacks on Xen and VMware." In: *IEEE Fourth International Conference on Big Data and Cloud Computing (BdCloud)*. 2014, pp. 737–744. DOI: 10.1109/BDCloud.2014.102.

[172] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors." In: *ACM Trans. Comput. Syst.* 15.4 (Nov. 1997), pp. 412–447. ISSN: 0734-2071. DOI: 10.1145/265924.265930.

[173] I. Eidus and H. Dickins. *Kernel Samepage Merging*. 2009. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html. (Online; accessed: 20.10.2020).

[174] C. A. Waldspurger. "Memory Resource Management in VMware ESX Server." In: *ACM SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 181–194. ISSN: 0163-5980. DOI: 10.1145/844128.844146.

[175] G. Miłoś, D. G. Murray, S. Hand, and M. A. Fetterman. "Satori: Enlightened page sharing." In: *Proceedings of 2009 USENIX Annual Technical Conference, USENIX*. 2009, pp. 1–14.

[176] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. "Difference Engine: Harnessing Memory Redundancy in Virtual Machines." In: *Communications of the ACM* 53.10 (Oct. 2010), pp. 85–93. ISSN: 0001-0782. DOI: 10.1145/1831407.1831429.

[177] D. Gullasch, E. Bangerter, and S. Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice." In: *IEEE Symposium on Security and Privacy (SP)*. 2011, pp. 490–505. DOI: 10.1109/SP.2011.22.

[178] Y. Yarom and K. Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7.

[179] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. "Wait a Minute! A fast, Cross-VM Attack on AES." In: *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014*. Springer International Publishing, 2014, pp. 299–319. ISBN: 978-3-319-11379-1. DOI: 10.1007/978-3-319-11379-1_15.

[180] Y. Azar, S. Kamara, I. Menache, M. Raykova, and B. Shepard. "Co-Location-Resistant Clouds." In: *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security CCSW '14*. ACM, 2014, pp. 9–20. ISBN: 978-1-4503-3239-2. DOI: 10.1145/2664168.2664179.

[181] Y. Han, J. Chan, T. Alpcan, and C. Leckie. "Using Virtual Machine Allocation Policies to Defend against Co-resident Attacks in Cloud Computing." In: *IEEE Transactions on Dependable and Secure Computing* PP.99 (2015), pp. 1–1. ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2429132.

[182] M. S. İnci, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. "Co-location Detection on the Cloud." In: *Constructive Side-Channel Analysis and Secure Design*. Springer International Publishing, 2016, pp. 19–34. ISBN: 978-3-319-43283-0.

[183] H. Hlavacs, T. Treutner, J.-P. Gelas, L. Lefevre, and A.-C. Orgerie. "Energy Consumption Side-Channel Attack at Virtual Machines in a Cloud." In: *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC)*. 2011, pp. 605–612. DOI: 10.1109/DASC.2011.110.

[184] N. Bin Sulaiman and H. Masuda. "Evaluation of a Secure Live Migration of Virtual Machines Using Ipsec Implementation." In: *IIAI 3rd International Conference on Advanced Applied Informatics (IIAIAAI)*. 2014, pp. 687–693. DOI: 10.1109/IIAI-AAI.2014.142.

[185] M. Aiash, G. Mapp, and O. Gemikonakli. "Secure Live Virtual Machines Migration: Issues and Solutions." In: *28th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2014, pp. 160–165. DOI: 10.1109/WAINA.2014.35.

[186] M. Anala, J. Shetty, and G. Shobha. "A framework for secure live migration of virtual machines." In: *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2013, pp. 243–248. DOI: 10.1109/ICACCI.2013.6637178.

[187] W. Fawaz, B. Daheb, O. Audouin, M. Du-Pond, and G. Pujolle. "Service level agreement and provisioning in optical networks." In: *IEEE Communications Magazine* 42.1 (2004), pp. 36–43. DOI: 10.1109/MCOM.2004.1262160.

[188] P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann. "Runtime prediction of service level agreement violations for composite services." In: *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*. Springer. 2010, pp. 176–186.

[189] T. Wood, E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. van der Merwe, and A. Venkataramani. "Disaster Recovery As a Cloud Service: Economic Benefits & Deployment Challenges." In: *Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing HotCloud'10*. USENIX Association, 2010, p. 8.

[190] N. Gonzalez, C. Miers, F. Redigolo, T. Carvalho, M. Simplicio, M. Naslund, and M. Pourzandi. "A Quantitative Analysis of Current Security Concerns and Solutions for Cloud Computing." In: *IEEE 3rd Int. Conf. on Cloud Computing Technology and Science CLOUDCOM '11*. IEEE CS, 2011, pp. 231–238.

[191] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya. "Cost of virtual machine live migration in clouds: A performance evaluation." In: *1st Int. Conf. on Cloud Computing (CloudCom 2009)*. Springer. 2009, pp. 254–265.

[192] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper. "Predicting the Performance of Virtual Machine Migration." In: *IEEE Int. Symp. on Modeling, Analysis Simulation of Comp. and Telecomm. Systems (MASCOTS)*. 2010, pp. 37–46. DOI: 10.1109/MASCOTS.2010.13.

[193] F. Salfner, P. Tröger, and M. Richly. "Dependable Estimation of Downtime for Virtual Machine Live Migration." In: *Int. J. on Advances in Systems and Measurements* 5 (1 2 2012).

[194] J. Lischka and H. Karl. "A virtual network mapping algorithm based on subgraph isomorphism detection." In: *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*. ACM, 2009, pp. 81–88. ISBN: 978-1-60558-595-6.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of April 14, 2022 (classicthesis v4.6).