ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering ($32^{th}$cycle)

# Formal assurance of security policies in automated network orchestration (SDN/NFV)

By

**Jalolliddin Yusupov**

******

**Supervisor(s):**
Prof. Riccardo Sisto
Prof. Adlen Ksentini, Co-Supervisor

Politecnico di Torino
2020

# Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Jalolliddin Yusupov
2020

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

# Acknowledgements

First and foremost I sincerely thank my supervisor, professor Riccardo Sisto, for his rigorous academic attitude, the selfless dedication of his time (including weekends) and efforts over the past three years that have allowed me to continuously improve and successfully finalize this dissertation. This work would not be possible also without the support, guidance and positive attitude of professor Guido Marchetto. I want to express my deepest appreciation and gratitude to my co-supervisor, professor Adlen Ksentini, for hosting me at EURECOM as a visiting researcher, for a valuable contribution to my work and for believing in me. I must also warmly thank Fulvio Valenza, whose support as part of his postdoctoral program encouraged me to overcome obstacles. In addition to being a great mentor, he became my best friend during the past years. I want to thank all the people in Lab9 for maintaining a good academic atmosphere. Finally, I thank my beloved family members and thank you for sharing the difficulties and achievements with me on this path, and for supplying me with unselfish love and support.

*Jaloliddin*

# Abstract

Nowadays, networks are evolving rapidly, and the need for more dynamic and automated network management is taking a significant role in defining the direction of this evolution. The new paradigms that are drastically changing the rules of the game are Software Defined Networking (SDN) and Network Function Virtualization (NFV). Despite the undeniable great benefits like scalability and flexibility they bring, these technologies also pose new issues regarding misconfigurations and security flaws since they rely upon external input to compose service graphs, to configure the virtual network functions (VNFs) and in order to enforce policies.

In this regard, traditional formal techniques have been proven to be a reliable means for verification, i.e. to discover such issues. However, the use of these techniques requires familiarity with mathematical foundations, where extensive training and expertise are required from network engineers to apply such methods. Other issues that have to be considered in virtualized networks are the need to use resources efficiently and the need to have automated procedures that let the administrators keep the pace with such rapidly evolving systems. The problem of efficiently placing the network functions across data centers can be solved by means of combinatorial approaches, but such approaches lack the formal verification part. In fact, in the literature, these two classes of problems have traditionally been addressed separately. A thorough search of the relevant literature yielded that reliable delivery of network services with a formal assurance about the network safety and security properties, providing at the same time efficient allocation of the resources and good automation, remains an open problem of greatest importance to be tackled in future research.

In this dissertation, we present our contributions to this field of research, which includes our proposed modeling framework and optimized verification and refinement technique. In particular, the modeling framework allows users to automatically

extract formal specifications from the source of network function applications, which is then converted into the compatible input format of different formal network analysis tools. In this way, we contribute to overcome the lack of familiarity with the formal notations. We utilize these formal models to formulate the joint virtual network placement and formal verification problem as a maximum satisfiability modulo theory problem (MaxSMT). Using this formulation, we target various instances of the problem in the cloud, 5G RAN, and industrial networks. In addition to the objectives discussed by the existing approaches, our model is able to encode more expressive constraints including the modeling of the forwarding behavior of the network functions, configuration parameters, and security policies. This approach, being formal, provides high confidence that the intended network security policies are correctly enforced.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

The design and implementation of software-centered, high-performance, and automation networks, as well as services and processes, enabled by a technological breakthrough, have become crucial to the future development of telecom operators around the world. The innovative trends of Network Function Virtualization (NFV) [2] and Software Defined Networking (SDN) [3] have opened up new business models, that allow telecom providers to lease or share their physical resources, and improve the flexibility and controllability of the network infrastructure. SDN divides the network control and data planes to provide (logically centralized) control plane programmability for novel networking virtualization (abstraction), simplified network (re)configuration, and policy enforcement. The NFV paradigm introduces a radical change in how the network architectures are being designed, splitting the bond between hardware and software, allowing for the development of network services as a software application. It has progressively gained attention in the industry in such a way that it is largely affecting how networks will be built in the future years.

In this context, the ETSI (European Telecommunications Standards Institute) Industry Specification Group (ISG) [4] is one of the main actors in defining a high-level functional architecture specification for NFV. With the close alignment of NFV and SDN as advanced technology areas, various other key groups such as 5G PPP[5], NGMN P1 WS1 5G security group[6], are concerned with standards and best practice for SDN, NFV, or often both. For instance, proposed specifications of the 5G PPP European research programme[5], state that SDN and NFV can coexist

separately, however, used jointly may introduce various advantages. This is why, 5G can greatly benefit from combined use of these technologies in delivering a rich set of services. NGMN P1 WS1 5G security group[6] focuses on the recent research in these technologies and discusses potential security challenges that must be addressed in order to ensure security of new 5G services. Design principles highlight various security requirements and recommendations, however they restrict the study of the assurance of security and service related aspects to inter-tenant/slice isolation. Other functional recommendations include requirements to be specified to prevent attacks by means of the forwarding plane[7] in case of reconnaissance attacks, DoS and resource exhaustion attacks.

However, architectural guidelines and standard mechanisms to develop virtual network functions (VNFs) are still missing in the research community. Therefore, service providers are bringing their own version of the VNFs to integrate into their own infrastructures. We emphasize that a generic model of the VNF founded on Topology and Orchestration Specification for Cloud Applications (TOSCA) has been introduced, which is in YAML Ain't Markup Language (YAML)[8]. However, it is meant only to define services of Cloud Applications, topology, and orchestration tasks. The concern is that the lack of these guidelines also to instantiate, configure, and operationalize the VNFs from various developers have a greater risk of causing potential network configuration errors. It, in effect, involves a tremendous amount of effort to provide the reliability, correctness, and security of the networks. For instance, VNF deployment experiences varius issues in reliability/availability management because of the software failures due to the security attacks. Any failure of VNFs in a particular service function chain (SFC) may lead to the suspension of the whole service. This is strongly related to the correctness of the behavior of the VNF defined in the source code. Checking the VNF software for bugs, verify its functionality, and refine the configurations if needed is one of the solutions that can be adopted in order to prevent those risks and errors. In this regard, there has been research progress recently in analyzing network correctness both in the data plane and control plane, with the help of formal methods[9–17].

Formal techniques accept as an input an abstract representation of the system in a tool-specific modeling language, targeted at a specific type of problem, and assign it formal semantics. It is commonly known that formal methods don't prevent errors or eliminate bugs that go below the formal model of the system. It means that incomplete or incorrect coverage analysis of the functionality results in not checking

all characteristics of the system required by the specifications. However, it is beyond the scope of this dissertation to investigate these well-known considerations. On the contrary, we first tackle the main limitations of the network function modeling to allow providers of NFV software to take advantage of formal analyses in virtualized networks. One of those limitations is the significant difference between the models written by network engineers and those allowed by current formal methods. For instance, existing tools such as SymNet [18], Alloy [19], VeriGraph [10], and NOD [13] have advanced in the last decade focusing on data-plane verification, but this difference in VNF definition might be a major barrier for their wide acceptance in real production environments. In principle, these formal engines are primarily designed using complex modeling procedures, require network functionality to be precisely modeled, mandates the user to utilize a certain kind of verification method, that requires specific technical skills, or oftentimes instructs developers to study a completely new language (e.g., Alloy in [19]).

We acknowledge this issue and aim to solve it with the help of well-known programming languages that are familiar to all programmers. After having introduced the necessary background in Chapter 2 and related work in Chapter 3, in Chapter 4 first we present a framework for a user-friendly network function modeling that network engineers can use to provide a formal model of their network functions to be used with formal methods for network verification. Moreover, by means of the generated models, we are able to formulate our next goal of joint verification and optimization.

As previously noted, NFV and SDN permit network operators to better utilize the network by mapping virtual components in optimized manner and delivering various network services at a reduced cost. This process is handled by the NFV orchestrator (NFVO) component. It is also in charge of coordination, instantiation, and configuration of VNFs. The complexity associated with the dynamic nature of the NFVO component may face several challenges on the path towards the goal of agile and dynamic service delivery. These challenges may include inconsistency, misconfiguration, absence of crash-freedom, bounded-execution[20], broken security tunnels, broken isolation or unreachability, which are, in many cases, tested with a simple ICMP Ping utility. The main purpose of our second goal is to present a methodology for optimal placement and formal verification of services to provide assurance that network-wide connectivity properties are always satisfied and that optimal allocation of resources is ensured before the delivery of the service. We

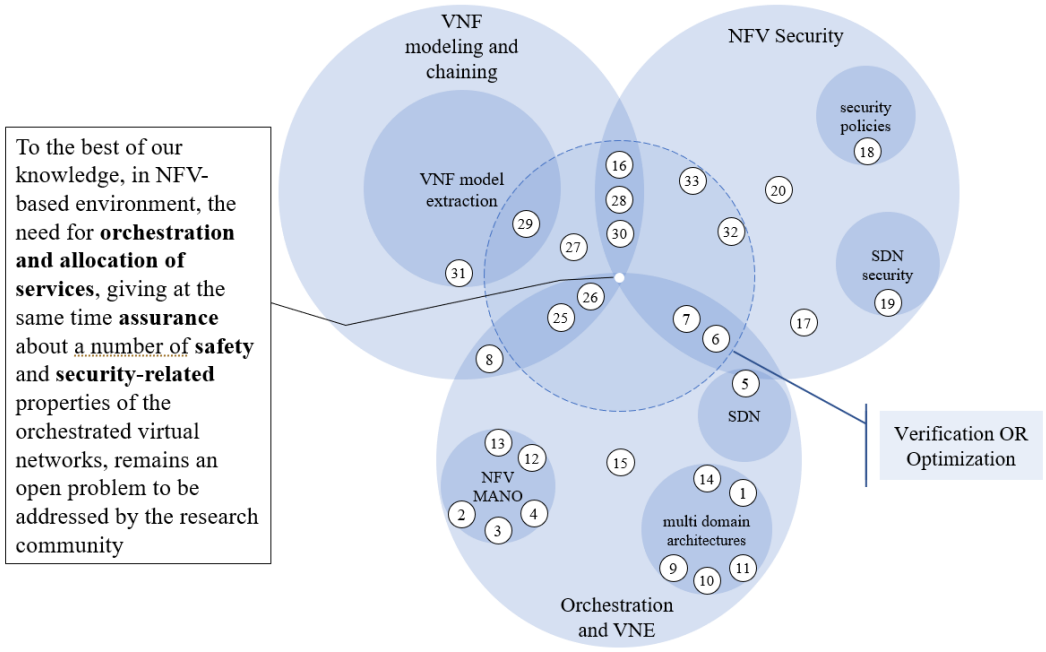


Fig. 1.1 Classification of the existing research activities.

believe that the delivery of reliable network services with an end-to-end service validation and assurance simplifies tasks for network administrators.

In Figure 1.1, we present the classification of the existing research activities and technologies we have collected during our research, with the help of four main circles. This is a Venn diagram that classifies research papers that are listed in Table 1.1. As indicated in the diagram, some of the items from the list concentrate on specific research areas, whereas the others cover a broader area of study. In the first circle, we listed the main research activities in VNF modeling and service functions chains. Then we analyzed existing literature that discussed the security aspects in NFV/SDN and grouped them in the second circle, which has a lot in common with the other circles. The third circle, combines works that tackle the problem of VNE during the orchestration phase. Lastly, the central fourth circle shows the research activities in the area of verification or optimization. In this literature review, we did not find studies on reliable delivery of virtual network services with an assurance about the safety and security properties, providing at the same time efficient allocation of the resources. In fact, efficient allocation alone does not assure that connectivity requirements mandated by the user are satisfied. Even if security property is a broad term, in this dissertation we use it referring more specifically to isolation and

Table 1.1 Taxonomy of surveyed techniques with corresponding publications

| | |
|---|---|
| 1 | Bernardos et al., (2016). 5GEx: Realising a Europe-wide multi-domain framework for software-defined infrastructures. Transactions on Emerging Telecommunications Technologies. |
| 2 | D. Chemodanov et al., (2019). A Constrained Shortest Path Scheme for Virtual Network Service Management. IEEE Transactions on Network and Service Management, |
| 3 | M. R. Rahman et al., (2013). SVNE: Survivable Virtual Network Embedding Algorithms for Network Virtualization. IEEE Transactions on Network and Service Management. |
| 4 | C. Aguilar-Fuster et al., (2018). Online Virtual Network Embedding Based on Virtual Links' Rate Requirements. IEEE Transactions on Network and Service Management |
| 5 | Bannour et.al., (2017). Distributed SDN Control: Survey, Taxonomy and Challenges. IEEE Communications Surveys & Tutorials. |
| 6 | V. Selis et al., (2019). A Classification-Based Algorithm to Detect Forged Embedded Machines in IoT Environments. IEEE Systems Journal |
| 7 | S. Spinoso et al., (2015) Formal Verification of Virtual Network Function Graphs in an SP-DevOps Context. European Conference on Service-Oriented and Cloud Computing |
| 8 | H. A. Alameddine et al., (2017). On the Interplay Between Network Function Mapping and Scheduling in VNF-Based Networks: A Column Generation Approach. IEEE Transactions on Network and Service Management |
| 9 | Iizawa et al., (2016). Multi-layer and Multi-domain Network Orchestration and Provision of Virtual Views to Users. COMPSAC |
| 10 | Guerzoni et.al., (2016). Analysis of end-to-end multi-domain management and orchestration frameworks for software defined infrastructures: An architectural survey. Transactions on Emerging Telecommunications Technologies. |
| 11 | Iizawa et al., (2016). Multi-layer and Multi-domain Network Orchestration and Provision of Virtual Views to Users. COMPSAC |
| 12 | H. Cao et al., (2019) A survey of embedding algorithm for virtual network embedding. China Communications. |
| 13 | M. A. Tahmasbi Nejad et al., (2018) vSPACE: VNF Simultaneous Placement, Admission Control and Embedding. IEEE Journal on Selected Areas in Communications |
| 14 | Kostas et.al., (2016). Multi-Domain Orchestration for NFV: Challenges and Research Directions. IEEE |
| 15 | R. Guerzoni et al., (2014). A novel approach to virtual networks embedding for SDN management and orchestration. IEEE Network Operations and Management Symposium (NOMS) |
| 16 | H. Wu et al., (2018). Virtualized Security Function Placement for Security Service Chaining in Cloud. IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS) |
| 17 | M. Pattaranantakul et al., (2018). NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures. IEEE Communications Surveys & Tutorials |
| 18 | C. Lorenz et al., (2017). An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement IEEE Communications Magazine |
| 19 | W. Ding et al., (2015). OpenSCaaS: an open service chain as a service platform toward the integration of SDN and NFV. IEEE Network |
| 20 | S. Lal et al., (2017) NFV: Security Threats and Best Practices. IEEE Communications Magazine |
| 21 | M. Peuster et al., (2017). Profile your chains, not functions: Automated network service profiling in DevOps environments. IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN) |
| 22 | M. Peuster et al., (2018). Understand Your Chains and Keep Your Deadlines: Introducing Time-constrained Profiling for NFV International Conference on Network and Service Management (CNSM) |
| 23 | J. Prados-Garzon et al., (2017) Analytical modeling for Virtualized Network Functions. IEEE International Conference on Communications Workshops (ICC Workshops) |
| 24 | M. Di Mauro et al., (2017). Service function chaining deployed in an NFV environment: An availability modeling. IEEE Conference on Standards for Communications and Networking (CSCN) |
| 25 | T. V. Phan et al., (2017). Optimizing resource allocation for elastic security VNFs in the SDNFV-enabled cloud computing. International Conference on Information Networking (ICOIN) |
| 26 | J. Pei et al., (2019). Efficiently Embedding Service Function Chains with Dynamic Virtual Network Function Placement in Geo-Distributed Cloud System. IEEE Transactions on Parallel and Distributed Systems |
| 27 | M. Baldi et al., (2018). Network Function Modeling and Performance Estimation. International Journal of Electrical & Computer Engineering |
| 28 | H. Gunleifsen et al., (2017) Security requirements for service function chaining isolation and encryption. IEEE 17th International Conference on Communication Technology (ICCT) |
| 29 | A. Gember-Jacobson et al., (2014). OpenNF: enabling innovation in network function control. SIGCOMM ACM |
| 30 | V. S. Reddy et al., (2016) Robust embedding of VNF/service chains with delay bounds. IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN) |
| 31 | A. Panda et al., (2014). Verifying Isolation Properties in the Presence of Middleboxes. CoRR |
| 32 | Shin et al., (2015). Verification for NFV-enabled network services. ICTC |
| 33 | I. Farris et al., (2019) A Survey on Emerging SDN and NFV Security Mechanisms for IoT Systems IEEE Communications Surveys & Tutorials |

reachability-based security. For instance, a security property may ensure the inability for an attacker located in a specific network location to reach a certain server. Other security issues, not addressed in this thesis are, for example, avoiding the possibility of attacks among VMs hosted on the same substrate[21], or avoiding that virtual resources are mapped on physical resources that do not comply with the security requirements of the virtual resource [22], or ensuring that the virtual networks of conflicting operators are mapped to different physical equipment.

As we mentioned earlier, only after the delivery of the service is considered done, safety or security requirements are generally checked by means of traditional network verification tools or testing tools. Because of this common, sequential way of operation, optimization and verification techniques were never addressed jointly in literature, which is evident because the intersection of the four circles shown in Figure 1.1 (the white circle in the middle) does not contain papers. Given this situation, we decided to position our research exactly at this intersection, because we believe that joining formal verification with optimal network function placement in the orchestration phase can bring benefits, by totally disallowing the actual delivery of network services that do not comply with the intended formal properties of such services.

In order to cover this gap, in Chapter 5 we present an approach addressing all the four fields presented in the circles, which merges the two main operations in one step. By means of the developed framework named Verifoo (Verification and Optimization Orchestrator), we generate an optimal allocation plan considering node and link parameters (e.g., CPU cycles, path length, delay, throughput, etc.) and provide a formal assurance of network security policies (e.g., reachability) to achieve embedding objectives under certain service constraints (e.g., sufficient bandwidth is available, and latency is within the recommended values). In general, this approach allows service providers to deliver more reliable and secure network services in an NFV infrastructure, providing assurance that the network properties are optimally and correctly enforced, in an automated fashion. Previous studies (e.g., [18, 10]) that were able to provide formal assurance of various network properties by means of formal verification of network behavior do not discuss the optimization objectives in formulating the verification problem. This dissertation fills this gap.

Lastly, in Chapter 6, we present an enhanced version of our framework named Verefoo (*Verified Refinement and Optimized Orchestration*), whose purposes go

beyond the verification and optimal placement of a user-provided service graph. Indeed, in this enhanced version we also address the extra problem of refining a given service graph and generating network function configurations automatically, in such a way that user-provided network security policies are satisfied and resource consumption is minimized. This enhancement not only prevents human errors, which would lead to potential deficiencies and conflicts during manual configuration phase, but also provides faster reaction when a misbehavior of the network is identified. In order to achieve this goal, we formulate this new joint problem in first-order logic as a MaxSMT instance, in order to provide high assurance about the correctness of the solution. As a result, we obtain an optimal placement and selection plan of network devices in a logical graph, and a minimized number of configurations of each network function. We also present the complete framework developed to integrate the proposed approached within well-known orchestrators from the industry in an automatic fashion. In particular, we first provide a northbound interface based on REST to the Kubernetes orchestrator (in charge of the orchestration of Dockers in a cloud scenario) to interact with our Verefoo framework. Secondly, we provide another approach of integrating Verefoo inside an OpenBaton orchestrator, which is an orchestration-centric implementation for cloud orchestration, as a plugin.

Chapter 7 finally concludes and provides direction for possible future improvements.

# Chapter 2

# Background

## 2.1 NFV and SDN in Telecommunications Service Provisioning

The Telecommunications Service Providers (TSPs) require their network to be highly secure, reliable and energy efficient. For this reason, traditional infrastructures consisted of dedicated hardware and closed systems. These closed systems often also integrate software counterparts in addition to hardware components. In spite of the introduction of new technologies and services, the revenue from traditional systems has drastically decreased, whereas the network usage has increased rapidly. The telecom industry saw the immense need for adoption of virtualization and cloud computing technologies after the introduction of "programmable networks". This embarked on a major movement on the generalization of equipment, fast service delivery, easier network updates, and reduced maintenance. Therefore, major telecommunications operators decided to introduce Network Function Virtualization (NFV). The main design concept of NFV is that the hardware uses unified industrialized servers. The virtual resource layer is deployed on the server to implement calls to the underlying hardware resources, and various network element software functions run on the standard server virtualization software. This in turn, provides flexible sharing and allocation of resources to improve resource utilization. Overall, NFV improves the concepts of the Telco over Cloud by delivering the virtualization of whole services, introducing virtual network functions.

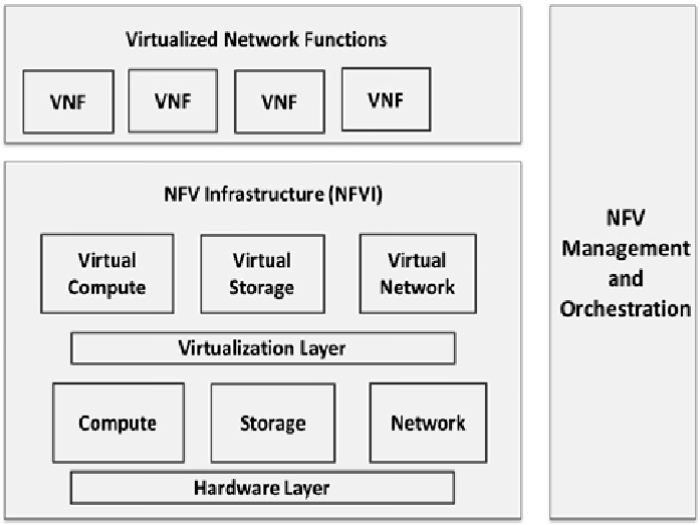


Fig. 2.1 An Overview of NFV Architecture and NFV Elements.

Currently, several standards organizations namely the European Telecommunications Standards Institute (ETSI)[23], and the 3rd Generation Partnership Project (3GPP)[24] are conducting research and developing NFV-related standards. In addition, there are various open source organizations dedicated to research and providing open source implementations of NFV open platforms, such as OPNFV[25], OpenStack[26], and OpenBaton[27].

**NFV architecture overview**

Figure 2.1 shows the main building blocks of the NFV, which was proposed by ETSI. The NFV system architecture includes: NFV infrastructure (NFV Infrastructure, NFVI), VNFs, NFV Management and Orchestration). Below we highlight the specific description of each component.

- The main function of NFVI is to provide resource pools for VNF deployment, management, and execution. NFVI needs to virtualize servers used for compute, network appliances, and storage hardware devices. NFVI can be deployed across multiple domains.
- VNF implements the functions of traditional hardware telecommunication network devices. This software implementation of the hardware devices is run on virtual machines (VMs).

- The NFV Management and Orchestration (MANO) mainly includes three parts: an NFV orchestrator (NFVO), a VNF manager (VNFM), and a virtual infrastructure manager (VIM).
  - NFVO is responsible for network services, orchestration and maintenance of physical / virtual resources and policies, and other maintenance management functions related to virtualization systems. It also allows to realize network service life cycle management, cooperate with VNFM to realize VNF life cycle management and global view function of resources.
  - VNFM implements the VNF Lifecycle Management (LCM) management of the VNF, including scaling, instantiation, and termination of VNF instances. It supports receiving the elastic scaling policy issued by NFVO to realize the elastic scaling of VNF.
  - VIM mainly handles the management, monitoring, fault reporting (for hardware resources), virtualization resources (i.e., at the infrastructure layer), and provides a virtualized resource pool for upper-level VNFM and NFVO.

**Software Defined Networking (SDN)**
SDN allows administrators to manage the entire network from a functional perspective. SDN distinguishes the control layer and the data layer of the underlying control traffic as separate entities, but maintains the connection within. Such separation provides more granular provisioning, more efficient service delivery and make it easier to manage. This also means that in the overall design of the network, flexible and intelligent control can be performed only at the management layer regardless of the underlying physical resources. SDN simplifies the most complicated and trivial tasks in the current network ecosystem, so it is only natural to be popular with data centers. In addition to this, SDN also has positive significance for other aspects such as the Internet of Things, 5G RAN, and Industry 4.0.

SDN and NFV technologies can be combined to deliver a more open, programmable and scalable networks, where SDN is in charge of the control plane and NFV delivers optimized deployment of the services. For example, there is a more complex and fragmented network system that has been extended to several data centers. In this case, it is more appropriate to adopt SDN to simplify the control of

network functions, traffic distribution, or automation. If the existing network environment is relatively uniform, but needs to implement specific network functions such as load balancing, one can consider using NFV to reduce overhead and complexity of hardware devices. In this thesis, we consider a mixture between network virtualization and SDN to deliver a more open, programmable and scalable network scenarios. For instance, to ease the process of defining and allocating Service Function Chains (SFCs).

## 2.2 Allocation of Services

Decoupling the hardware devices yields the issue of how service graphs are realized by the infrastructure network. We need to emphasize that the infrastructure network can be virtualized too. The problem of allocating service graphs in an infrastructure network is one of the optimization challenges of an NFV environment and is referred as the Virtual Network Embedding (VNE)[28] problem. VNE solves the problem of allocating network services provided by the user. Upon arrival of a service request, an optimal allocation plan must be found for network nodes in a subtrate network, to achieve objectives under various constraints (e.g., required storage, bandwidth, latency). Solving VNE is known to be NP-hard[29]. Additionally, the complexity of VNE increases if we need to introduce additional constraints about the verification of the network properties, which will be discussed later in this thesis.

## 2.3 Formal Methods and Formal Verification

*Formal methods* are techniques based on mathematical foundations, which are suitable for specifying, developing, and verifying computer systems. Applying formal methods to system design is intented to improve the correctness, and hence the dependability, of the design using appropriate mathematical analysis, like in other engineering disciplines. More specifically, *formal verification* is the way of using mathematical methods to prove the correctness or inaccuracies in the design process of computer hardware and software systems, according to some form or specification.

### 2.3.1 Verigraph

NFV and SDN introduce new issues including the complexity, vendor compatibility, maintaining network security and ordering of network and security functions. This ordering of network functions, allows to prevent conflicts, inconsistencies or sub-optimizations, especially considering the cases where multi-domain administration is involved. Currently, most VNFs require a manual input by network administrators, in order to select specific fields for various low-level configurations.In the context of telco operators, manual configuration may involve multiple iterations if there is a misconfiguration in the initial phase. This approach, besides requiring human intervention and being costly, may require dramatic measures depending on the size of the network. To address these issues, Verigraph[10] proposes a formal method relying on network modelling that is then used to verify the satisfiability of reachability-based properties (named policies) against a network composed of both stateless and stateful network functions. This work is based on the following features: (i) reachability checking relies not only on topology constraints, but also on the configuration of each function in the network (e.g. rules for firewall, blacklists for antispam functions, packet values for end-host); (ii) the approach has a high usability because it supports a number of possible network functions so that telco operators or administrators can describe and in turn verify a variety of network topologies; (iii) in case of network topology changes, the modelling approach is totally reusable because it separates the generic forwarding principles from the functional behavior of the existing network functions.

Verigraph turns the model and the policies into First-Order Logic (FOL), utilizes a solver to obtain a formal assurance that the correctness of the network behavior is maintained with respect to the policies. FOL is a formal system, namely in the form of symbolic reasoning system, also known as quantificational logic and "Predicate logic", although these statements are not precise enough. We should note that the first-order logic is different compared to a basic “propositional logic”, as it uses variables that take values over various domains and quantifiers, such as: $\exists$ y (exists a value y ); $\forall$ y (for all the values of y ).

### 2.3.2 Satisfiability problem

The Satisfiability (SAT) problem in computer science and mathematical logic is the problem of finding whether there is an assignment of the boolean variables of a set of propositional logic formulas that makes all such formulas true. In 1971, Cook [30] proved an early NP-completeness result of the satisfiability problem.The SAT problem was also the first of this kind to be proven. At first, researchers focused on the application of SAT in hardware testing, circuit verification and other fields. The development of SAT solvers has found a significant importance in the research of associated works in the area of Electronic Design Automation (EDA). Then SAT was widely used and applied to various emerging fields, such as static program analysis, test case generation, etc. SAT is only oriented to propositional logic formulas, and for this reason its expression ability has high limitations.

Due to these shortcomings, researchers have extended SAT to SMT. Satisfiability Modulo Theories (SMT) is oriented to first-order logic formulas. It adds reasoning about propositional logic and has stronger expressive power. SMT incorporates a variety of background theories, and propositional variables in the formula. It can be a theoretical formula that can directly describe the high-level information of the problem. For example, SMT's array theory can directly describe array definitions and related operations. In practical applications, SMT is not limited to a single theory, usually a combination of multiple theories is used.

### 2.3.3 Maximum Satisfiability Modulo Theories

When using SMT, very often, we would like to find not only the verification result but, if the verification succeeds, also obtain the best conditions under which this can happen, according to some optimization objective. In that instance, the verification problem becomes also an optimization problem. Optimization problems usually are modeled as linear programming models. However, translation of a set of first order logic (FOL) formulas to a linear programming problem is not an easy task and, in most cases, impracticable. Another option is to turn an SMT problem into an optimization problem by putting a constraint on an objective function and doing binary search to find the minimum value of such constraint that does not result in UNSAT. This allows to keep the problem formulated in a logical language and provide it to an appropriate solver, capable of jointly performing verification of the

logic formulas, and optimization. In this dissertation, we are concerned with this approach, using MaxSMT tools. The MaxSMT problem is the optimization version of the Satisfiability Modulo Theories (SMT) problem. Both SMT and MaxSMT are represented by means of propositional logic formulas in the Conjunctive Normal Form. As we have stated earlier, the aim of the SAT and SMT problems is to determine a truth assignment that satisfies all of the clauses; otherwise, they return an unsatisfiability report. At this point, if there is no satisfying model, we still want to know if there are subsets of clauses that are satisfied, and find the one where as many as possible are satisfied. This is the idea of the MaxSMT problem. The clause is a conjunction of literals. In MaxSMT, each literals is assigned with a weight, and the goal is to solve the optimization problem of minimizing the weight of falsified collection of clauses. In this kind of problem, the clauses are characterized as hard or soft clauses. Hard closes are the ones that must be satisfied while soft clauses may or may not be satisfied. There are various forms of MaxSMT problem and we adopt Weighted Partial MaxSMT variation, in this thesis. In contrast to other variations, it allows us to assign weights to soft clauses, where hard clauses must be satisfied and weights of the satisfied clauses must be maximized.

### 2.3.4 SMT/MaxSMT solvers

The specific implementations of SMT and MaxSMT solving technologies are called SMT and MaxSMT solvers. z3 is a new and efficient theorem prover from Microsoft Research that receives as input sets of FOL formulas. Under the hood, z3 is a SMT solver. It can resolve satisfaction problems and thus formalize an approach to constraint programming. An extension of z3 is z3Opt, which can solve the MaxSMT problem in those scenarios in which arbitrary models are not enough, and applications want to minimize or maximize values. As z3 is an engine developed to be used by any other tools, it provides several APIs that can be exploited by various tools.

### 2.3.5 MaxSMT example using Z3

In this section, we provide an example to demonstrate the use of Z3 API. We encode a placement problem using Java API to find the optimal locations of the VNFs in a substrate network. Let us assume that there are two substrate nodes with 65 GB and

100 GB available storage to host VNFs. Each VNF requires 5GB, 10 GB and 50GB available storage to be hosted in a substrate node. Daily financial costs associated with deploying VNFs on those nodes are fixed: 100 USD and 20 USD respectively. Our goal is to minimize the number of substrate nodes in use and minimize the embedding cost (i.e., financial cost ).

To represent the placement problem in Z3, we introduce boolean variables $y_j$ and $x_{ij}$ that take true value when substrate node $j$ is in use and when VNF $i$ is hosted on substrate node $j$, respectively. A detailed description of the API can be found here[1].

```
1 //required variables
2 Context ctx = new Context();
3 Optimize mkOptimize = ctx.mkOptimize();
4 BoolExpr x11,x12,x21,x22,x31,x32;
5 BoolExpr y1,y2;
```

We introduce a $bin()$ function to represent these boolean variables as integer/real variables. For our example, the sum of all storage required by VNFs allocated on a substrate node should be less than or equal to the storage available on that substrate node and it is expressed as:

```
1  //first substrate node constraints
2  ArithExpr leftSide = ctx.mkAdd(
3      ctx.mkMul(ctx.mkInt(5), bin(x11)),
4      ctx.mkMul(ctx.mkInt(10), bin(x21)),
5      ctx.mkMul(ctx.mkInt(50), bin(x31)));
6  mkOptimize.Add(ctx.mkLe(leftSide, ctx.mkMul(ctx.mkInt(65), bin(y1))));
7  //second substrate node constraints
8  ArithExpr leftSide = ctx.mkAdd(
9      ctx.mkMul(ctx.mkInt(5), bin(x12)),
10     ctx.mkMul(ctx.mkInt(10), bin(x22)),
11     ctx.mkMul(ctx.mkInt(50), bin(x32)));
12 mkOptimize.Add(ctx.mkLe(leftSide, ctx.mkMul(ctx.mkInt(100), bin(y2))));
```

In addition to these inequalities, we need to represent explicitly that each VNF can be mapped onto exactly one node. For our example, such constraints take the following form:

```
1 mkOptimize.Add(ctx.mkEq(ctx.mkAdd(bin(x11),bin(x12)), ctx.mkInt(1)));
2 mkOptimize.Add(ctx.mkEq(ctx.mkAdd(bin(x21),bin(x22)), ctx.mkInt(1)));
3 mkOptimize.Add(ctx.mkEq(ctx.mkAdd(bin(x31),bin(x32)), ctx.mkInt(1)));
```

Finally, when substrate node is in use, there is at least one VNF deployed on this node and for our example we have:

[1] https://z3prover.github.io/api

```
mkOptimize.Add(ctx.mkOr(
ctx.mkImplies(y1, x11), ctx.mkImplies(y1, x21), ctx.mkImplies(y1, x31)));
mkOptimize.Add(ctx.mkOr(
ctx.mkImplies(y2, x12), ctx.mkImplies(y2, x22), ctx.mkImplies(y2, x32)));
```

We use `AssertSoft` function to minimize the weighted sum of constraints for the same ID. It is important to use the negation of the variable if there is a need for minimization.

```
mkOptimize.AssertSoft(ctx.mkNot(y1), 100, "new");
mkOptimize.AssertSoft(ctx.mkNot(y2), 20, "new");
```

By feeding these objectives along with the formulas defined so far to the Z3 MaxSAT solver, we obtain, if possible, a model that satisfies all hard clauses, including the ones about capability, while minimizing the number of nodes in use. In the case of our example, the solver says the model is satisfiable with value true given to the following variables: x12, x22, x32, y2. We can conclude from the output that all VNFs are placed at substrate node 2 with the cost of 20.

# Chapter 3

# Related work

This chapter surveys previous work related to formal assurance of security policies in automated network orchestration and discusses in detail how our solution advances the state of the art. First, we review network function modeling in developing a formal model of network functions for network verification. Then limitations are listed, which we have faced while observing the different network function models designed for formal methods. Second, we survey existing literature, which employs these models in deciding the optimal placement of network functions. We compare these solutions to ours from several perspectives, and show how our solution provides a joint optimization and verification for the most current solutions. Finally, we discuss works in automated network orchestration. We narrow our discussion to automatic security mechanisms for firewalls.

## 3.1 Network function modeling

There is currently no standard and well-known modeling language that can be used to precisely characterize the forwarding behavior of network functions. Existing literature on modeling network functions is mostly concentrated on data plane or control plane verification and each tool requires input models written in its own language. In this section, related work of the field is discussed, then limitations are listed, which we have faced while observing the different network function models designed for formal methods.

### 3.1.1 Simplicity of models

Modeling of network functions is effective for various uses, ranging from finding scalability issues in applications to finding network configuration bugs, especially with the use of formal verification tools. On the other hand, former modeling of network functionalities is challenging and requires a detailed understanding of the specific verification tool internals, semantics, and modeling language.

With this problem in mind, the introduction of an automated approach to generate models eliminates the necessity of having detailed knowledge in the formal verification domain and helps engineers to quickly determine the behavior of services comprising different types of network functions, starting from a more user-friendly description of the involved network functions. Notice that it is well-known how formal methods do not prevent possible errors in constructing the formal model. In other words, incomplete or wrong models may lead to errors in the verification process. However, the investigation of these well-known issues is out of the scope of our study. Instead, given the fact that formal methods are a widely accepted methodology for property verification, we address the challenge of giving to NFV software providers the possibility to construct formal models by means of a programming paradigm they are familiar with. This would significantly lower entry barriers to these powerful verification approaches, somehow also reducing the probability of introducing errors in the model with respect to the utilization of complex formal languages.

Imperative languages such as Java, Python, C++ focus on describing *how* a program operates. A network function developer can write a code that describes in exact detail the forwarding decisions that the network function must make when a packet is received from one of its interfaces as a sequence of steps, without having the complexity of a function implementation. On the contrary, declarative languages [31, 32] used in logic-based formal verification tools do not specify a step or sequence of steps to execute, but rather predicates that must hold. The conceptual gap between these two paradigms is the vital challenge solved by our approach.

### 3.1.2 Support of stateless and stateful functions

The existing verification methods can be divided into two groups according to their ability to model stateful functions. The former group focuses on modeling the forwarding behavior of stateless devices(e.g., switches and routers [33, 34], ACL (Access Control List) Firewall [35], simple load-balancer[14]); by this we mean that the behavior is not modified until the control plane explicitly changes the configuration and there is no record of previous interactions. The latter group[36–38] also considers the devices that are dynamic, in which every packet that the network device receives may alter the internal state, and the output is dependent on the sequence of previously encountered packets. Considering the fact that a significant portion of network devices are stateful, such as learning firewalls, load balancers, intrusion detection systems and the like, one cannot ignore these devices when verifying network configurations. Thus, we propose a general template to model, that covers a wide range of network functions, including both stateless and stateful ones.

### 3.1.3 State of the art

There are two categories of related work to be considered. The first category relies on the analysis of programming languages source code. In the past few years, there has been a significant amount of work done to provide a proper support for the translation of programming language sources to the input models of verification tools. Some of these tools are for example Bandera [39] and JavaPathFinder [40]. Both of them are founded on model checking, and the output of the tools are related to a generic computer programs (i.e., they do not extract models of network functions). The main target of these models is the identification of programming errors and bugs. In contrast to these works, we consider only the forwarding behavior of network functions, and we are not interested in all the details of the network function's code execution. We also want formal verification of network configurations to be extremely fast because it has to be performed in real-time when a network change occurs. For this reason, we need to extract customized, domain-specific models.

A proposal more similar to our target is NFactor [41], which provides a solution to automatically analyze the source code of a given network function to generate an abstract forwarding model, motivated by network verification applications. While

relying on advanced tools([42]) and methods ([43–45]) from the field of code analysis, there is no requirement on the structure of the written code. This feature is considered as an advantage from a generality point of view. SymTCP[46] is built on top of the popular concolic execution engine S2E[47] and analyzed vulnerabilities in source code of deep packet inspection (DPI) systems for protocol ambiguities. It proposed automated ways to construct packets that can successfully de-synchronize the state of a DPI middlebox from that of a (end) server.

Unfortunately, creating a model that captures all code paths of a network function is challenging because the state processing may be hidden deep in the code. This may cause the analysis to miss certain state changes. For example, implementations might use pointer arithmetic to access state variables, which is difficult to trace, and solutions based on concolic testing do not deal with these language features appropriately. Another limitation of the approaches based on extraction of models from source code is that the code of many network functions is proprietary, hence source code is not available for them.

Another category of approaches and methods for static network analysis is based on hand-written function models ([13, 18, 12, 10]). For instance, Network Optimized Datalog [13] requires a Datalog (declarative logic programming language that syntactically is a subset of Prolog) both for network and function models and policy specifications. BUZZ [12] relies on manually written models of network functions defined in a domain-specific language. As stated above, modeling network functionality for using these tools is challenging and requires a detailed understanding of the verification tool semantics. Our automated approach to generate models aims at eliminating the necessity of having detailed domain knowledge and helps network engineers to quickly determine the behavior of a network function.

On the contrary, SymNet [18] describes models using an imperative, modeling language, known as Symbolic Execution Friendly Language (SEFL). While the way this language has been designed has similarities with the modeling technique we propose, this approach lacks the idea of ease of modeling, by introducing a new language. Even though the authors provide parsers to automatically generate SEFL models from real network functions, this generation only covers routers and switches. Our approach, instead, relies on the well-known user-friendly programming languages and can be used to describe any kind of virtual network function.

## 3.2 Formally verified network function placement

The classical literature on Virtual Network Embedding (VNE) is based on the Integer Programming (IP) formulation of the problem of optimally mapping VNF instances to specific nodes and links in the substrate network, but it does not take into consideration reachability analysis during optimization. A number of NFV placement or orchestration frameworks have been proposed in the literature. Most of them, such as PACE [48], propose smart VM placement to deploy VNFs without considering reachability properties at all, so they cannot optimize or control the way packets are forwarded. Other state-of-the-art approaches, such as APPLE [49] and VTVSynth[50], also consider reachability policies while providing VNF placement, where policies describe the sequence of VNFs that each class of flow needs to traverse in order. However, these approaches only assure that traffic is forwarded by the SDN switches according to the policies while they do not provide formal assurance that reachability policies will really hold because they do not use precise models of the forwarding behavior of middleboxes.

In contrast to traditional methods for solving the VNE problem based on integer programming and heuristic methods, which are limited to a set of constraints over binary, integer, or real variables, the problem addressed in this chapter is to allocate VNFs while also formally checking that the desired reachability policies will hold, considering formal models of the forwarding behavior of all the VNFs involved, including their configurations. Even though the transformation of propositional calculus statements into integer and mixed-integer programs is possible [51], combinatorial encoding is impractical in most cases, and we were often not able to generate MaxSAT encodings for many of the instances when using it. The reason is that the formulation we have adopted in this thesis is the first-order logic - an extension of propositional logic that covers variables for individual objects, quantifiers, symbols for functions, and symbols for relations.

How to take security properties into account when deploying SGs is another important consideration. During our research, we observed existing approaches ([52, 53]) on the problem of security-aware optimal VNE. These approaches make an assumption that each virtual network request has a set of security requirements. These requirements only comprise constraints on the confidentiality levels of the substrate nodes and isolation of the resources. In terms of security services, authentication, data integrity, confidentiality, and replay protection requirements should be provided

[54],even if the delivery of these requirements depends on the specific configuration and implementation details. On the other hand, several VNFs (e.g., NAT) can modify or update packet headers and payloads. In these environments, it is difficult to protect the integrity of flows traversing such VNFs and reason about reachability properties without using precise behavioral models of VNFs.

## 3.3 VErified REFinement and Optimized Orchestration

### 3.3.1 Network Security Automation

Automatic security mechanisms have been proposed for firewalls in literature. [55–57] define policy-based automatic methodologies to configure firewalls with respect of some security requirements; however, they lack formal assurance of the correctness and they are designed to work in traditional networks with hardware appliances, instead of an NFV or cloud environment. Instead, [58–60] enable automation when fixing firewall misconfigurations, exploiting formal verification at the same time; nevertheless, these approaches do not allow the creation of the firewall policies from scratch and, in the case of [58] and [59], they are not still thought for virtualized networks.

On the contrary, automatic configuration of other NSFs has been investigated less extensively. [61] represents the most important work for a policy refinement activity that is not limited to a single kind of function; however, it does not contribute to the creation of a complex security Service Graph, since it is limited to the configuration of function chains.

Then, about automatic service composition, [62–65] contribute to create network services in cloud environments by exploiting novel networking technologies, but without the support of formal verification. Other works [66] [67] establish the optimal firewall allocation in a virtualized service according to some cost functions; even though their purpose is similar to the allocation feature of Verefoo, however, they do not focus on a larger pool of network functions among which to choose for enforcing the security policies, neither automatically define their configuration.

### 3.3.2 NFV and cloud orchestration

The MANO of an NFV and cloud infrastructure plays a central role in modern computer networks. Centralizing the deployment, management and monitoring of the *VNFs*,simplifies the way the service providers reach their users. Our focus in this thesis is to provide additional capabilities to the existing MANO solutions by integrating the framework we presented in this thesis. We have analyzed various types of cloud MANO platforms on the market and observed that none of them provide the automatic configuration of VNFs by solving the verification and optimization problems in one step. We have selected the most popular and widely used projects that conform to the guidelines of NFV MANO:

- *Open Source MANO (OSM)* [1] is the prominent framework, proposed by ETSI, whose design currently inspires all the alternative tools for orchestration of VNFs in a virtualized environment. Since it is an open source project, it must be able to interface with a number of functions or platforms that belong to different vendors. For this reason, a research trend is to define novel architectural models which can abstract from the vendor-specific peculiarities of each VNF implementation: examples of this work are modelling languages such as TOSCA and YANG. [4] OSM offers a multiple *VIM* support, which means it could be used with multiple Infrastructure-as-a-Service technologies (e.g. OpenStack, AWS, OpenVIM, VMware) for resource orchestration. It allows also to combine them with different SDN Controller technologies (e.g. ONOS, floodlight, ODL) to manage the underlying connectivity. A monitoring system and an experimental support for SG are provided as well. In order to manage the VNFs instance OSM adopts Juju of Canonical as VNF Manager.

- *Open Baton* [2] is an NFV MANO solution compliant with the ETSI NFV MANO specifications. It has a modular architecture, in which a message broker (RabbitMQ) grants the communication between a set of different orchestration and supplementary services. The main services offered are the orchestration of resources, the monitoring system and a set of drivers, which allows using this platform over multiple VIM technologies. In order to extend Open Baton for supporting other VIMs, one has to create a new driver plug-in and a specific

[1] Open Source MANO. [Online]. Available: https://osm.etsi.org
[2] Open Baton. [Online]. Available: https://openbaton.github.io

VNFM for this technology. This approach gives an interesting flexibility to VIM support. Indeed, Open Baton has been the first NFV platform to give support to Docker Engine as VIM. SG mechanisms.

- *OpenStack Tacker* [3] is an additional service for the OpenStack framework [4]. This service provides NFV Orchestration functionalities (e.g. VNFs/NSs management), leveraging the services included in the OpenStack IaaS platform (e.g. Neutron, Nova, Heat). The main advantage of using this platform is the full support for SFC with the service named networking-sfc and SGs; in the *Network Service Descriptor (NSD)* we could provide an high-level description of FSPs and Classifiers as well.

Other works proposed in the literature on NFV and cloud orchestration, that are worth mentioning, are [68] and [69]. On one side, vConductor [68] is a framework that can construct and monitor virtual enterprise networks with a multi-objective resource scheduling of the VNFs. On the other side, [69] describes a model-based approach that exploits *network function-agnostic* software components such as translators and gateways to install functional configurations into each network function of a complete service that is managed with a framework integrated in a cloud management platform. These frameworks are richer in terms of security functions that can be scheduled, but an automatic policy-based configuration is not integrated, thus requiring a human contribution in the creation of the virtual service; in fact, also [69] only performs a translation of vendor-independent rules instead of a policy refinement. Finally, [70] proposes a modular NFV architecture that permits policy-based management of VNFs, handling their whole life-cycle and exploiting an Information Model to provide an abstraction of network resources, network control functions and VNFs capabilities; however, their limitation is that the only network security capability that is modelled is access control.

[3] Tacker. [Online]. Available: https://docs.openstack.org/tacker/latest/
[4] OpenStack. [Online]. Available: https://www.openstack.org

# Chapter 4

# Network function modeling for verification purposes

## 4.1 Problem description and contributions

Considering that almost any NFV developer can introduce sophisticated VNF software, there is a greater chance of software bug and network configuration errors. In this regard, a great deal of work is required to preserve safety, security, and the correctness of the network. For this reason, verification of networks is the main factor in eliminating faults and building reliable systems. In this regard, traditional formal methods are believed to be mature and well suited for verifying numerous network properties in various circumstances ([13, 71, 72]). Those techniques make use of formal models defined in a specific language and exploit model checking, theorem proving or SAT solvers to provide formal assurance. This in turn, challenges developers of VNF software to either acquire the knowledge of formal modeling or translate the existing code of the network functions into a specific format accepted by those verification tools. To avoid these difficulties, we introduce a novel framework consisting of a library, parser, and translator. By means of the typical set of high-level operations provided by the library, a developer is able to describe the forwarding behavior of a generic network function in a well-known language. Even if we have no experimental evidence that it is easier than learning a modeling language, we believe an automated approach to generate models eliminates the necessity of having detailed knowledge in the formal verification domain and helps engineers to quickly

determine the behavior of a VNF-based network, starting from a more user-friendly description of the involved VNFs.

A base class definition of a generic VNF is provided by the framework. This base class can be easily extended by inheriting basic properties, methods, and data types, which allows to customize the behavior of the function and develop various types of specific network functions. The other element of the framework is a parser, which is in charge of compiling and analyzing the source code, and extract a parse tree from it. We represent the parse tree in the form of the platform-independent XML specification, which represents an abstract formal model of a VNF. This XML definition may then be converted into various models accepted by different verification tools. The fairly generic intermediate model generated by the parser includes all the parameters of the network function behavior, but many of those parameters are not necessarily used by all verification tools. In such cases, our translator component designed for each specific verification tool will omit the parameters not used by that tool from the output model.

In this thesis, we utilize the network verification tools SymNet [18] and VeriGraph [10] as use cases and Java as a specification language. Despite this, we want to emphasize that the methodology introduced in this chapter can be represented by means of various computer programming languages, and due to the level of abstraction of the intermediate parsed model, we can develop translators even for other verification tools of network forwarding behavior. First, we provide literature overview in Section 3.2. Then, we present the approach and the main components of the framework in Section 3.3, which includes the definition of the Library, Parser and Translator. Finally we provide the evaluation of the proposed framework.

## 4.2 User-friendly verification oriented modeling approach

A library, a parser, and a translator are the main components of the proposed framework. First, we present our library and introduce the main design guidelines in modeling network functions by means of the imperative programming paradigm. For the sake of completeness, we also show examples of VNFs from our rich network function catalog. Next, we describe the fundamental aspect of the parser, which

allows us to extract an abstract model of a network function from the source code definition. In general, the parser accepts as an input the source code of the VNF written respecting the guidelines of the library. Finally, the translator component of the framework converts the intermediate abstract model into a tool specific representation, which would be hard to obtain manually. As a matter of fact, we provide separate translators for each type of verification tool in order to be compliant with the language used.

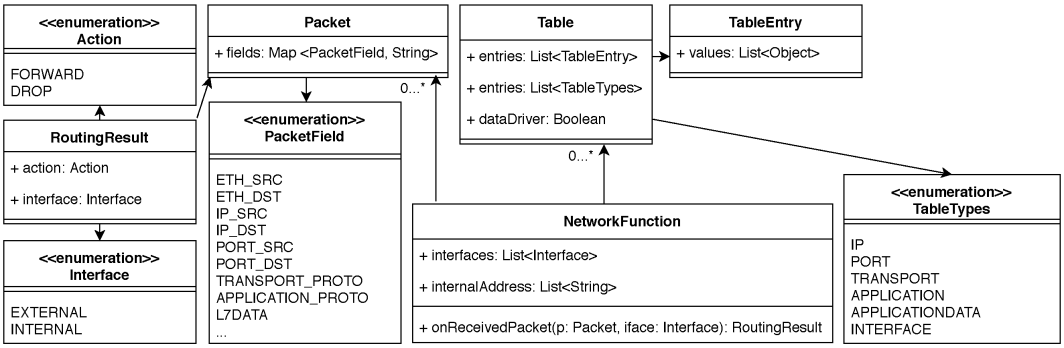


Fig. 4.1 Class diagram of the Library.

### 4.2.1 Library: Network function catalog

In Figure 4.1 we present a static structure of the library in terms of a class diagram notation, where the nodes are classes, and the edges are dependencies or generalizations. It allows a developer to understand the basic building blocks in defining the NF behavior. Users may simply define forwarding decisions of the network function in accordance with these design principles. The developer is required to instantiate objects of the library classes and use available methods that match specific behavior of individual NFs. The operations include forwarding, blocking or duplicating an IP packet, matching a specific field of a packet from a table, keeping track of packets and others. This can be achieved by using the basic syntax of the Java language.

Firstly, for any network function, it is required to create a class that extends the library class `NetworkFunction`, which includes generic behavior of a network function. `NetworkFunction` is an abstract class whose abstract methods have to be implemented in the concrete extension provided by the user. The source code that is shown in Listing 4.1 is an example for an ACL (Access Control List) enabled stateless firewall. The main method of the class (`onReceivedPacket()`)

accepts two parameters. The first one represents the incoming packet object while the second one specifies the interface on which the function has received the packet. The actions that can be inserted inside `onReceivedPacket()` are divided into the following categories: instructions to get, set or check the contents of a packet field, instructions to store and retrieve a value into/from a lookup table defined by the user, instructions to define the forwarding action to be performed on a packet (through a `RoutingResult`). `RoutingResult` is the return value type of the `onReceivedPacket()` method. It expresses the routing decision of the network function, after processing of the incoming packet. Its constructor receives the following input parameters:

- A packet object that the network function produces.
- The action to perform on this packet (forward or drop).
- The forwarding direction (i.e. the interface the packet is forwarded to in case of forward action).

Considering the fact that the model describes the forwarding behavior of the VNF, we suppose that the forwarding operation involves a single packet in response to a received packet. Even if this restriction could be removed by updating the `RoutingResult` definition, for all the functions we considered it was not an issue. Moreover, we aim to provide the assessment of the possibility or impossibility for a function to deliver a certain type of packets on an interface, rather than representing exactly how many packets of a certain type it can send for each received packet.

The `Interface` class provides means to define which logical interface packets are sent to or received from by a VNF. A typical need when writing network function models is to differentiate various sets of interfaces. For instance, a NAT (Network Address Translator) network function separates the network into two areas (an internal area and an external one) and applies different rules on the incoming packets if they are received from the internal or the external interface. Incoming packets from the external interface are forwarded to their destination only if a connection already exists while packets from the internal interface are always forwarded unless the NAT runs out of ports. Similarly, most of the network functions such as firewall, NAT, and others differentiate the interface to which packets can be transferred, from the interface from which packets arrive. We will refer to them as external and internal interfaces, respectively, in this thesis.

```
1  public class AclFirewall extends NetworkFunction {
2      private Table aclTable;
3      public AclFirewall() {
4          super(new ArrayList < Interface > ());
5          this.aclTable = new Table(2, 0);
6          this.aclTable.setTypes(Table.TableTypes.Ip, Table.TableTypes.Ip);
7      }
8      @Override
9      public RoutingResult onReceivedPacket(Packet packet, Interface iface) {
10         if (iface.isInternal()) {
11             TableEntry entry = aclTable.matchEntry(packet.getField(PacketField.
                   IP_SRC), packet.getField(PacketField.IP_DST));
12             if (entry != null) return new RoutingResult(Action.DROP, null, null);
13             return new RoutingResult(Action.FORWARD, packet, externalInterface);
14         } else {
15             TableEntry entry = aclTable.matchEntry(packet.getField(PacketField.
                   IP_SRC), packet.getField(PacketField.IP_DST));
16             if (entry != null) return new RoutingResult(Action.DROP, null, null);
17             return new RoutingResult(Action.FORWARD, packet, internalInterface);
18         }
19     }
20 }
```

Listing 4.1 Behavior of the ACL firewall represented in an imperative form.

The `Packet` class models an IP packet and it gives access to the fields of the IP header that are relevant for the forwarding behavior of the network function. As it can be seen from Figure 4.1 the `enum` type `PCKField` contains the constants representing the packet header fields that can be modelled. Moreover, our framework can support TCP and application protocols. For the moment, only HTTP and POP3 are supported for demonstrative purposes. This list can be extended indefinitely to include more protocols when necessary. It is worth noting that the `clone()` method of the class is convenient in case of packet modifications. This, in turn, allows modifying the "clone" in the meantime keeping the old packet unchanged.

The `Table` class models a typical lookup table as a collection of `TableEntry` objects stored by the network function. For example, in the case of NAT-enabled network function shown in Listing 4.2, an object of this class stores the pairs of source/destination IP address/port of open connections. The code of the NAT VNF consists of three rules due to the three forwarding actions. To represent the explicit state change that occurs in the stateful VNF, $natTable.storeEntry(e)$ method invocation is used. In contrast to the matchEntry() method, used to model a table lookup for the stateless VNF, here the user models a condition stating that there is a packet received, looking at the VNF's internal state.

Table 4.1 List of supported commands, instructions and operators

| Data types: int, String, boolean, byte |
|---|
| Boolean operators |
| Comparison operators |
| Statements: return, if, if-else,<br>variable assignment, method invocation |

Alternatively, a table object can be used in an ACL firewall to store port numbers or IP addresses to be blocked. The `matchEntry()` method of the class, on the other hand, serves to retrieve an entry from the table matching the value of the object. The `TableEntry` class itself contains a list of objects whose size is set according to the integer that the constructor receives. Moreover, it is necessary to define the expected field type the table stores. This helps the parser to observe the type of entry being retrieved from the table and extract the model accordingly. For instance, the element for IP source address and destination addresses is stored as an IP data type in the XML notation of the model. The list of currently supported types is given in `enum` type `TableTypes` (see Figure 4.1).

To grant the second objective of our work and to offer greater flexibility, developers are provided with a sufficient level of freedom to follow the guidelines and "skeleton" of the model mentioned above as building blocks of complex network functions. In particular, there is no restriction on the number of tables, conditions, and on the actions that follow these conditions, on the order of packet operations and the programming language used to describe these models. However, our framework supports only a subset of the object-oriented programming language features, which are supported by C++, Java, Objective-C, and others. The list of supported features is shown in Table 4.1. While we believe this subset is enough for our purposes, it can be extended in order to further improve the user experience when specifying network functions.

### 4.2.2 Parser: Operating principles of the parser

The parser performs a syntactic analysis of the source code conforming to the rules of a formal grammar. Next, it builds an in-memory parse tree for easy translation into another language describing the network function behavior and then converts the tree into XML notation. This process is made of the following sequence of steps:

- the identification of the instructions in the code that lead to a packet being sent through an interface;
- the identification of the instructions related to state insertion (write) and retrieval (read);
- the identification of the conditions (IF statements) that are traversed to reach the above-mentioned instructions.

Finally, the analysis lets us identify (i) the possible sequences of instructions that can modify and finally send a packet, with all the conditions under which each one of them can be executed and (ii) the sequences of instructions that can lead to a modification of the state of the network function, and all the conditions under which each alteration can occur.

The implementation of the parser takes advantage of *Abstract Syntax Tree (AST)*, which is a tree representation of the abstract syntactic structure of source code. AST is provided by the compilers of all well-known programming languages like C++, Java, JavaScript, and Python. The formal model generated by the parser takes a form that is motivated by the vision of OpenFlow [73] forwarding abstraction of the form <*match, action*>. This abstraction model has been borrowed from the existing modeling techniques [41, 15] and most of the verification tools of forwarding behavior ([10, 16, 18]) rely on the models of this abstraction.

The parser generates the model in an XML format that expresses this abstraction, by building the <*match, action*> pairs from the parsed AST. We classify the boolean conditions in the code that lead to a return statement or to a state change statement as "match" and the forwarding and state changes as "action" respectively. The "match" conditions are further classified into two categories: those that refer to state variables and those that are state independent. Similarly, the "action" rules are categorized as: those that trigger a state transition and those that trigger a forwarding action. They are referred to as "state" and "flow" respectively, as shown in Table 4.2. This categorization is needed in the translation phase of the model, because the verification tool-specific model representation is formed depending on the type of <*match, action*> tuples. In the following subsections, we walk through the steps the parser takes in building the abstraction model for stateless and stateful network functions and cover the classification of rules in detail.

Table 4.2 Intermediate output of the parser for ACL firewall

| | | | |
|---|---|---|---|
| 1 | Match | Flow | recv(p_1,INTERNAL) |
| 2 | | | !matchEntry(p_1.IP_SRC, p_1.IP_DST) |
| 3 | | | p_0.IP_SRC == p_1.IP_SRC |
| 4 | | | p_0.IP_DST == p_1.IP_DST |
| 5 | | | p_0.PORT_SRC == p_1.PORT_SRC |
| 6 | | | p_0.PORT_DST == p_1.PORT_DST |
| 7 | | | p_0.TRANSPORT_P == p_1.TRANSPORT_P |
| 8 | | | p_0.APPLICATION_P == p_1.APPLICATION_P |
| 9 | | | p_0.L7DATA == p_1.L7DATA |
| 0 | | State | * |
| 1 | Action | Flow | send(p_0,EXTERNAL) |
| 2 | | State | * |

```
1  @Override
2  public RoutingResult onReceivedPacket(Packet packet, Interface iface) {
3      Packet packet_in = null;
4      packet_in = packet.clone();
5      if (iface.isInternal()) {
6          TableEntry entry = natTable.matchEntry(packet_in.getField(IP_SRC),
7              packet_in.getField(PORT_SRC));
8          if (entry != null) {
9              packet_in.setField(IP_SRC, (String) entry.getValue(4));
10             packet_in.setField(PORT_SRC, (String) entry.getValue(5));
11
12             return new RoutingResult(Action.FORWARD, packet_in, extInterface);
13         } else {
14             TableEntry e = new TableEntry(7);
15             e.setValue(0, packet_in.getField(IP_SRC));
16             e.setValue(1, packet_in.getField(PORT_SRC));
17             e.setValue(2, packet_in.getField(IP_DST));
18             e.setValue(3, packet_in.getField(PORT_DST));
19             e.setValue(4, natIp);
20             e.setValue(5, String.valueOf(new_port));
21             e.setValue(6, new Date());
22             packet_in.setField(IP_SRC, natIp);
23             packet_in.setField(PORT_SRC, String.valueOf(new_port));
24             natTable.storeEntry(e);
25             return new RoutingResult(Action.FORWARD, packet_in, extInterface);
26         }
27     } else {
28         TableEntry entry = natTable.matchEntry(
29             packet_in.getField(IP_SRC), packet_in.getField(PORT_SRC),
30             packet_in.getField(IP_DST), packet_in.getField(PORT_DST));
31         if (entry == null)
32             return new RoutingResult(Action.DROP, null, null);
33         packet_in.setField(IP_DST, (String) entry.getValue(0));
34         packet_in.setField(PORT_DST, (String) entry.getValue(1));
35         return new RoutingResult(Action.FORWARD, packet_in, intInterface);
36     }
37 }
```

Listing 4.2 Behavior of the NAT in an imperative form.

### Stateless network function

As an example, we use the NF definition for an ACL firewall by means of our library, which is listed in Listing 4.1. This function uses a `Table` object named `aclTable`. The method invocation in line 6 specifies that the table has two columns, both of type IP address (`Table.TableTypes.Ip`). The parser will store this information in the XML notation of the intermediate model. This table acts as a "blacklist": if the source and destination IP addresses of the received packet match an entry in the table, a drop action is performed. The illustration of such abstraction for the stateless ACL firewall network function obtained automatically from the simple source code definition, is presented in Table 4.2, where only the first rule extracted from the code is presented. In fact, for each forwarding action in the `onReceivedPacket()` method, the parser constructs a separate rule. The forwarding actions are recognized by the parser by looking for the presence of `RoutingResult` class instance creation in return statements, where the action argument is equal to `Action.FORWARD`. For instance, the method in Figure 4.1 leads to the generation of two distinct rules due to the presence of two forwarding actions (lines 13 and 17, Figure 4.1). Due to the stateless nature of the NF, the final model does not contain any information regarding the state (i.e., lines 10 and 12 in Table 4.2 are empty).

The parser proceeds backward in the control graph of the code, starting from the selected forwarding action until it reaches the entry point of the method. In this way, it obtains the sequence of statements that have to be executed in order to reach the selected forwarding action. From this sequence, the parser extracts the conditions that must be satisfied in order for the sequence to be executed. They are essentially the conditions of the if statements found in the sequence, plus an additional predicate taking the form `recv(p,i)`, where $p$ and $i$ are the packet and interface passed to `onReceivedPacket()`. This last predicate expresses the condition that packet $p$ can be received on interface $i$. In all these conditions, variables are substituted with the values assigned to them in the sequence of statements, explicitly or implicitly. Due to the nature of network functions that we consider, *onReceivedPacket*() function do not interface with other definitions/instantiations for handling packet forwarding, thus avoiding cases of recursive functions.

Table 4.3 Intermediate output of the parser for NAT (part 1)

| | | | |
|---|---|---|---|
| 1 | Match | Flow | recv(p_1,INTERNAL) |
| 2 | | | p_0.IP_DST == p_1.IP_DST |
| 3 | | | p_0.TRANSPORT_P == p_1.TRANSPORT_P |
| 4 | | | p_0.APPLICATION_P == p_1.APPLICATION_P |
| 5 | | | p_0.L7DATA == p_1.L7DATA |
| 6 | | State | p_0.IP_SRC == p1.IP_SRC |
| 7 | | | p_0.PORT_SRC == p1.PORT_SRC |
| 8 | | | recv(p_2,INTERNAL) |
| 9 | | | p_1.IP_SRC == p_2.IP_SRC |
| 10 | | | p_1.PORT_SRC == p_2.PORT_SRC |
| 11 | Action | Flow | send(p_0,EXTERNAL) |
| 12 | | State | * |

### Stateful network function

Models of stateful NFs are more complex, because the match field may regard not only packet flows but also states, and the action to be performed not only may forward the packets but also may trigger an update on state components. In order to explain the difference between the two cases, we analyze the outcome of the parser for a NAT network function. From Figure 4.2, it is easy to realize that the model for the NAT network function consists of three rules due to the three forwarding actions. To extract these rules, the parser proceeds in an analogous manner to that shown in section 4.2.2.

At the beginning the parser classifies this as a stateful network function due to the explicit state change that occurs in the `natTable.storeEntry(e)` method invocation, in line 26, and identifies `natTable` as part of the function state. In contrast to the `matchEntry()` method, used to model a table lookup for the stateless network function, here the parser classifies the `matchEntry()` method invocation listed in line 7 as a state specific conditional statement, which corresponds to lines 6-10 in Table 4.3. In order to differentiate the received packet from the modified packet to be sent and from the packet received in the past, we use three variables $p\_1$, $p\_0$ and $p\_2$. Additionally, the packet modifications done by the NAT result in a form of constraints, where the source IP and port addresses of the new packet, that is being sent, must correspond to the values retrieved from the entry, as shown in lines 6,7 of Table 4.3. Whereas the flow-related, state-independent conditions in lines 1-5 of Table 4.3 state that the new packet that is being forwarded, must keep the rest of the fields of the received packet unchanged. This is the first

Table 4.4 Intermediate output of the parser for NAT (part 2)

| | | | |
|---|---|---|---|
| 1 | Match | Flow | recv(p_1,INTERNAL) |
| 2 | | | p_0.IP_DST == p_1.IP_DST |
| 3 | | | p_0.TRANSPORT_P == p_1.TRANSPORT_P |
| 4 | | | p_0.APPLICATION_P == p_1.APPLICATION_P |
| 5 | | | p_0.L7DATA == p_1.L7DATA |
| 6 | | State | p_0.IP_SRC == "natIP" |
| 7 | | | p_0.PORT_SRC == "new_port" |
| 8 | | | recv(p_2,INTERNAL) |
| 9 | | | !(p_1.IP_SRC == p_2.IP_SRC) \|\| |
| 10 | | | !(p_1.PORT_SRC == p_2.PORT_SRC) |
| 11 | Action | Flow | send(p_0,EXTERNAL) |
| 12 | | State | store(p0,"new_port") |

rule the parser builds following the if branches in lines 6 and 9 (Figure 4.2). When the parser visits the else branch of the code that starts at line 14, it extracts the set of conditions that lead to another forwarding action and builds the rule shown in Table 4.4. In contrast to the first rule, the second rule contains an action that triggers a state transition, by storing the new entry in the internal state of the network function (line 12 in Table 4.4). This implies that the NAT translation table does not contain an entry matching the IP and port source addresses as given in lines 9,10 (of Table 4.4) and the new entry is inserted in the translation table of the NAT network function.

The final rule to be extracted covers the behavior of the NAT network function when receiving a packet from the external interface, which is extracted in a similar manner. This approach combines network-level abstractions and network function-level abstractions that, together, make the verification task possible. In other words, we abstract away (i) the order of packets; (ii)the relationship between the states of different network functions; and (iii) the relationship between states of different packets within each network function. The last two abstractions are inspired by [74, 75]. The primary limitation of this approach is that the state information related to a specific network function does not reflect the dynamic data plane elements.

### 4.2.3 Translator: Support for tools that verify forwarding behavior of the network

One of the strengths of our approach is the ability to serialize the final model abstraction across different languages, thus being able to target different verification

programs. For instance, VeriGraph[76] exploits network function models expressed as formulas in First Order Logic (FOL) [77], taking the form

```
1 send(NF, destination, packet) -> CONDITIONS
```

The main functions that model operational behaviors in a network are:

- *Bool send(node_src, node_dest, packet)* which returns true if source node *node_src* can send packet *packet* towards destination node *node_dest*;
- *Bool recv(node_src, node_dest, packet)* which returns true if destination node *node_dest* can receive packet *packet* from source node *node_src*.

The right hand side of the formula expresses the conditions under which the packet is forwarded. These formulas are difficult to write. Hence, VeriGraph can greatly benefit from the automatic generation of models.

The conditions that are included in each rule are combined in conjunctive normal form (CNF) in order to obtain a single "match" and "action" rule respectively. For example, the rule in Table 4.4 results in the following FOL formula for VeriGraph:

```
1  ((send(n_Nat,n_0,p_0) && !(isInternal(p_0.IP_DST))) ->
2  E(n_1, p_1 |(recv(n_1,n_Nat,p_1) &&
3  isInternal(p_1.IP_SRC) &&
4  !(E(n_2, p_2 |(recv(n_2,n_Nat,p_2) &&
5  isInternal(p_2.IP_SRC) &&
6  (p_1.IP_SRC == p_2.IP_SRC) &&
7  (p_1.PORT_SRC == p_2.PORT_SRC))))&&
8  (p_0.IP_SRC == natIp) &&(p_0.PORT_SRC == new_port) &&
9  (p_0.IP_DST == p_1.IP_DST) && (p_0.PORT_DST == p_1.PORT_DST) &&
10 (p_0.TRANSPORT_P == p_1.TRANSPORT_P) &&
11 (p_0.APPLICATION_PROTOCOL == p_1.APPLICATION_PROTOCOL) &&
12 (p_0.L7DATA == p_1.L7DATA))))
```

Line 1 of this formula is interpreted as that a new packet $p_0$ can be sent to a node $n_0$ through the external interface of the network function, which is translated as a negation of `isInternal(p_0.IP_DST)` VeriGraph specific predicate.
This send action can occur, only if it is not possible to receive another packet $p_2$, having the same port and source IP addresses as packet $p_1$. The rest of the model is translated as follows. However, the state transition primitive in line 12 does not take place in the final translation because VeriGraph does not support data structures to keep track of the internal state of the network function. Despite this, VeriGraph can model any stateful function that depends not just on static forwarding rules, but also on the sequence of previously encountered packets, introducing multiple implications in the function model.

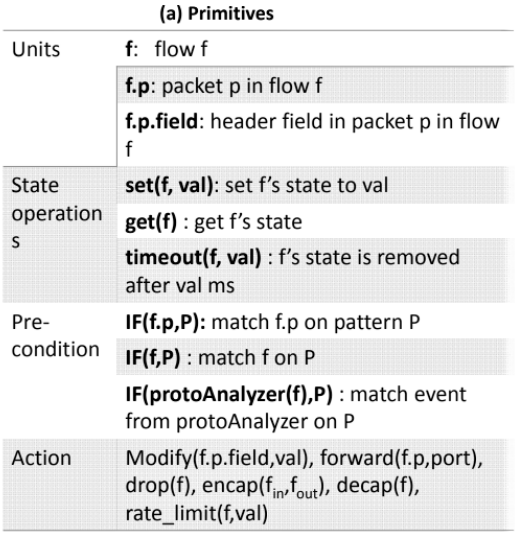


Fig. 4.2 SFC-Checker model primitives.

On the other hand, SFC-Checker[15] supports predicates explicitly altering the state of the network function in the form of temporal forwarding behavior using Finite State Machines (FSM). Table 4.5 shows the output of the translator for the NAT network function model rule given in Table 4.4, by means of SFC-Checker supported primitives shown in Figure 4.2. As evident from the table, the state independent condition in line 1 of Table 4.4 is translated using the *pre-condition primitive* `IF` introduced in SFC-Checker. Whereas the state-relative conditions in lines 6-10 of Table 4.4 take the form of *state operations* primitive - `get()`. By means of the `set(f.p,"new_port")` operation, the information related to the state change in the NAT table is delivered. This operation is triggered when the translator matches the double equal sign on the packet $p\_0$, which in fact, the packet to be sent. It is important to note that the distinction present in the XML representation is not reflected to this translation and the translator produces a single packet $f.p$ for all three $p\_0, p\_1, p\_2$ packets.

Table 4.5 Representation of the NAT model in SFC-Checker format

| Match(f,s) | Action |
|---|---|
| IF(f.p.src,INTERNAL),<br>get(f.p.ip_src)!=f.p.ip_src&&<br>get(f.p.port_dst)!=f.p.port_dst | forward(f.p,out)<br>set(f.p,"new_port")<br>Modify(f.p.ip_src,"nat_ip")<br>Modify(f.p.port_src,"new_port") |

| *Instruction* | *Description* |
|---|---|
| `Allocate(v[,s,m])` | Allocates new stack for variable v, of size s. If v is a string, the allocation is handled as metadata and the optional m parameter controls its visibility: it can be global (default) or local to the current module. If v is an integer it is allocated in the packet header at the given address; size is mandatory. |
| `Deallocate(v[,s])` | Destroys the topmost stack of variable v; if provided, the size s is checked against the allocated size of v. The execution path fails when the sizes differ or there is no stack allocated for variable v. |
| `Assign(v,e)` | Symbolically evaluates expression e and assigns the result to variable v. All constraints applying to variable v in the current execution path are cleared. |
| `CreateTag(t,e)` | Creates tag t and sets its value e, where e must evaluate to a concrete integer value. |
| `DestroyTag(t)` | Destroys tag t. |
| `Constrain(v,cond)` | Ensures that variable v always satisfies expression cond. The execution path fails if it doesn't. |
| `Fail(msg)` | Stops the current path and prints message msg to the console. |
| `If (cond,i1,i2)` | Two execution paths are created; the first one executes i1 as long as cond holds. the second path executes i2 as long as the negation of cond holds. |
| `For (v in regex,instr)` | Binds v to all map keys that match regex and executes instruction instr for each match. |
| `Forward(i)` | Forwards this packet to output port $i$. |
| `Fork(i1,i2,i3,...)` | Duplicates the packet and forwards a copy to each output port $i1, i2, ...$ |
| `InstructionBlock(i,...)` | Groups a number of instructions that are executed in order. |
| `NoOp` | Does nothing. |

Fig. 4.3 SEFL instruction set.

Similarly, the Symbolic Execution Friendly Language (SEFL) proposed by SymNet [18] requires models in the *<match, action>* formalism and replaces unknown values in the conditions with symbolic values. This helps SymNet to explore different paths of the model. Eventually, the output of the translation for the NAT models in Table 4.3 and Table 4.4 is identical to the model demonstrated in SymNet [18]:

```
1   InputPort(0):
2   Constrain(IPProto,==6) //only do TCP
3   Allocate("orig-ip",32,local)
4   Allocate("orig-port",16,local)
5   Allocate("new-ip",32,local)
6   Allocate("new-port",16,local)
7   Assign("orig-ip",IpSrc) //save initial addr
8   Assign("orig-port",TCPSrc) //save initial port
9   Assign(IpSrc,"...") //perform mapping
10  Assign(TcpSrc, SymbolicValue())
11  Assign("new-ip",IpSrc) //save assigned addr
12  Assign("new-port",TcpSrc) //save assigned port
13  Forward(OutputPort(0))
```

It is important to note that SymNet injects only one packet per execution when it performs verification [78]. Hence, we cannot translate the conditional statements on multiple packets of our data-driven network functions such as NAT and Web Cache into SEFL language. As a result, models generated for SymNet cannot take state into account. In addition, in those network functions the idea of private or public network is described in terms of specific ports, as there is no such network division in SymNet. However, by defining a new abstraction, we can generate a model of a stateful function specifically designed for SymNet only. As our goal is to provide a

Table 4.6 Elapsed time to parse network function models.

| Network function model | Time to parse (ms) |
|---|---|
| Mail Server | 862 |
| IDS | 869 |
| NAT | 880 |
| Web Server | 920 |
| Web Cache | 952 |
| Firewall | 957 |
| Antispam | 963 |

generic modeling language that supports most of the verification tools, this approach is out of scope for this work.

## 4.3 Implementation and evaluation

The resulting models, describing the forwarding behavior, are well suited for verification of the basic network invariants such as reachability and isolation. Among the existing logic-based verification tools we selected VeriGraph[10] and SymNet[18] as use cases to show how the model generated by the parser can be exploited, after proper translation, by a real verification tool. VeriGraph and SymNet are formal verification tools that can automatically verify networks by checking certain policies before the service deployment. In this context, the term network is used to indicate a chain of network functions such as (load balancer, antispam, packet filter, DPI and so on) that starts from a source node and ends into a different destination node. In response to a verification request, a model of the network and the involved network functions with their configurations, is checked against the provided policies, for instance isolation properties between multiple devices in the network.

To automatically validate network connectivity policies at scale, both the verification engines of VeriGraph and SymNet exploit an off-the-shelf SAT solver (Z3), which checks the satisfiability of constraints over various theories. The logical formulas in this case define the policies to be verified, and the behavior and configuration parameters of each network function.

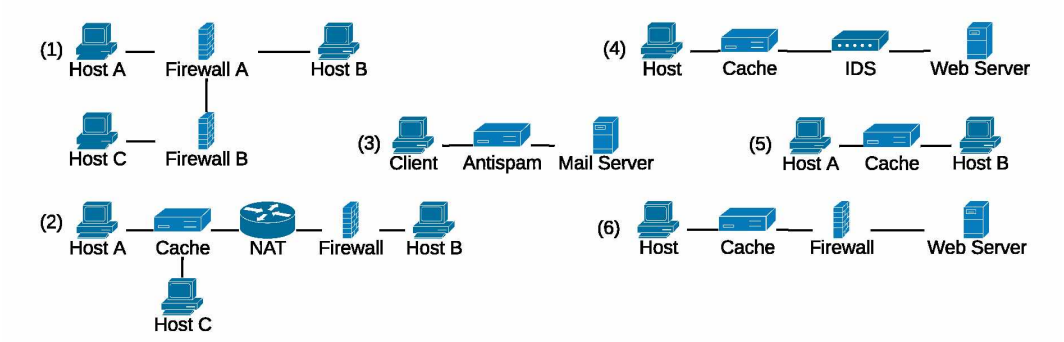


Fig. 4.4 Example topologies generated for our simulations.

In order to check the correctness of the generated final models, we constructed a set of experiments with different network topologies comprising the catalog of functions, and we performed a number of custom tests on the selected verification tools. Verigraph and SymNet can perform different kinds of verification tests: reachability, which consists of checking if at least one packet can arrive at the destination from the source node, and isolation, namely, that packets sent from one host (or class of hosts) can never reach another host (or class of hosts).

Figure 4.4 illustrates the set of topologies adopted for our tests. By means of these tests, we show that generated abstract models are close to the actual behavior of network functions and can be used in various scenarios. For instance, topology (1) involves two firewalls and three end hosts. Firewalls are configured according to the following policies:

1A. Firewall A denies all traffic between host A and C and the default action of the firewall is allow.

1B. Firewall B denies all traffic between host B and C and the default action of the firewall is allow.

This test includes two isolation properties to be checked. In particular, we consider two packets, one flowing from host A to host C, and another one flowing from host A to host B. Taking into account the above firewall policies, we expect the isolation property is satisfied in case of A-to-C, indicating that no packet can reach the host C from A, while we expect it is not satisfied in the case of A-to-B. The other test cases are set up as follows (the numbers refer to the corresponding topology in Figure 4.4):

2. Rule: firewall denies traffic between NAT and host B. B is located in private network where NAT table contains an entry related to the previous connection B-to-A. Property: isolation between hosts A and B. Action: send a packet from host A to B.

3. Rule: antispam performs an application layer content filtering. A packet containing a string "discount" in its body is blocked. Property: isolation between mail server and client. Action: send a packet containing a string "discount" in its body from client to mail server.

4. Rule: DPI drops a packet containing a specific string in the body of the packet. Property: isolation between host and web server. Action: send a packet containing the specific string in the body from host to web server.

5. Rule: local storage of web cache contains *"www.google.it"* URL address. Property: isolation between host A and B. Action: send a web page request containing *"www.google.com"* URL from host A to B.

6. Rule: firewall denies traffic between host and web server. Local storage of web cache is empty. Property: isolation between host and cloud web server. Action: send a packet from host to web server.

It is also possible to consider more complex, real life scenario, where IDS contains an entry in its "blacklist" table with not allowed function code equal to 43. Any specific type of packet not containing a function code equal to 43 expected to be allowed.

Table 4.7 delivers the results we obtained implementing these categories of tests in VeriGraph and SymNet. In order to exploit the models generated by our framework it is required to construct actual service requests forming a chain or graph by connecting them together with links. Verigraph then uses a different approach to verify complex service graphs by making the use of chain extraction. This allows Verigraph to verify a service request comparatively faster than SymNet. In terms of computation time Verigraph performed more than twice faster compared to the SymNet verification tool in all scenarios. In the table, 'SAT' means the isolation property is satisfied, while 'UNSAT' means that the isolation property is not satisfied. Comparing the test results obtained by using a set of manually written models and

the ones obtained by means of the automatically generated ones (starting from the high-level description and then generated using the parser), we found that results are identical, as expected. We emphasize that the time required to generate these models never exceeded one second. Whereas it would require a significant amount of time from a developer to define hand-written models. This confirms the correctness of our modeling approach and also shows the efficiency of the developed framework.

Table 4.7 Execution time of VeriGraph and SymNet compared. Column N represents the number of the corresponding topology illustrated in Figure 4.4.

| **N** | **Scenario** | **Verification results** | **Time to verify autogenerated NF model (ms)** | |
|---|---|---|---|---|
| | | | VeriGraph | SymNet |
| (1) | DoubleFwTest A | SAT | 214 | 530 |
| | DoubleFwTest B | UNSAT | | |
| (2) | CacheNatFwTest | SAT | 318 | 787 |
| (3) | AntispamTest | SAT | 275 | 681 |
| (4) | DPITest | SAT | 192 | 475 |
| (5) | CacheTest | UNSAT | 260 | 644 |
| (6) | CacheFwTest | SAT | 200 | 495 |

## 4.4 Discussion

In this chapter, we described a "user-friendly" approach to network function modeling for formal verification of forwarding behavior. We focus on breaking the barrier between the two ways of representing a network function: the imperative-centric function definition (proper of network function developers) and the higher-level declarative representation (used by formal verification experts in order to instruct logic-based verification tools). The proposed approach provides a method for NFV developers to translate from the former to the latter automatically. The method relies on the modeling technique, which includes a modeling library, a parser, and a translator. NFV developers who write the actual implementation of the network function can use the framework to provide a formal description of their functions to be used in a verification process. Instead of modeling every detail of a VNF, they can only focus on the forwarding behavior, in order to enable the formal verification

of typical reachability oriented properties (e.g. isolation or absence of forwarding loops). This approach tends to be affordable, since owner of the source code has a full understanding of the function forwarding behavior.

We validated the correctness of the models obtained using our framework by means of different verification tools.

Considering what are the current requests of the market and looking at the possible future developments, this framework presents a further step towards the real implementation of these new concepts inside the networks. In fact, the proposed framework and the available verification tools may be a basic structure to define Virtual Network Functions and test the overall network functionality before deployment.

# Chapter 5

# Formally verified network function placement

## 5.1 Problem definition and contributions

The NFV paradigm leads to the virtualization of services and applications that, in traditional networks, run on proprietary hardware appliances (e.g., firewall, DPI, etc.). This abstraction allows a more flexible network in which the network functions that, in this context, are called VNF, can be simply moved from one server to another one. Moreover, a series of network functions can be chained together to offer more complex use cases. The use of virtualized applications in place of physical hardware also enables administrators to take network functions in and out of service and to scale them up and down quickly. The NFV paradigm is based on the usage of an orchestrator, which decouples the instances of the network functions from the requested virtual service. As we have stated in Chapter 1, even though the orchestrator provides a simple way to deliver or release a service composed of a series of VNFs, it does not provide verification for the correctness of the whole service, while providing an optimal placement plan (i.e., VNE). For example, it does not check if two endpoints are indeed reachable in the service in solving the VNE problem, as this would require the formulations of how the different VNFs work. Even though the NFV automation approach offers undeniable great benefits like scalability, flexibility, and optimization, on the other hand, it poses new issues regarding misconfigurations and security flaws since it relies upon external tools

in verifying network properties. This problem can be prominent, especially by introducing other objectives in addition to verifying the absence of misconfiguration, such as efficient utilization of resources that host services in physical topology, where the management and orchestration of the system could reach a high level of complexity. In spite of wide coverage of the data plane and control plane verification in the literature, they are usually performed after or before an NFV orchestrator delivers enforcement of service requests in the infrastructure. However, addressing the problem of VNE and verification separately can cause problems. For example, if verification is done after VNE and deployment, misconfiguration can be discovered only after the actual deployment of the service, which can be risky in some cases (e.g., a firewall that lets a dangerous flow pass). In the other scenario, an orchestrator that receives a verified SG could perform decomposition and then deployment, respecting the capabilities of the infrastructure, which might be different from the initial layout. This new actual mapping requires a new verification of the lower layer network configurations and the same verification process with a different set of VNFs. In case of a policy violation, the orchestrator must reiterate through the alternative solutions and again perform the verification.

To address this range of issues, we present a framework named Verifoo (Verification and Optimization Orchestrator), an extension of Verigraph that is capable of performing joint optimization and verification. It follows the recent trend for MANO[4] of many platforms, adding the integration of formal verification with the placement procedure and thus formally verifying that services work before their actual deployment. Verifoo can deploy requested SGs by searching from a shared catalog of resources. Since multiple mappings of an SG onto an infrastructure are possible, the orchestrator component also performs an optimal placement on the basis of given performance parameters. It delivers a formal assurance of reachability-related policies that are relevant for safety and security. It requires, as an input, an SG, the configurations of the VNFs in the SG, the physical topology, and a set of network policies to be verified.

In this chapter, we provide three variations of our methodology, focusing, respectively, on typical network infrastructures, on industrial control systems, and on radio access networks. Before describing the methodology used in Verifoo, in section 3.2 the related literature on verification and optimal VNE is reviewed. Then, in Section 5.4.2, the definition of the problem is given in formal terms, considering an example of embedding a Service Function Chain (SFC) on a substrate network topology. In

Table 5.1 Summary of key notations

| Symbols | Notations |
|---|---|
| $N^s, E^s, L^s$ | Set of substrate nodes/endpoints/links |
| $N^v, E^v, L^v$ | Set of VNFs/enpoints/links to be allocated |
| $A^s_N, A^s_L$ | Attributes of substrate nodes/links |
| $A^v_N$ | Attributes of virtual functions |
| $l^s_{j,k}$ | Link between substrate nodes indexed by j and k |
| $n^v_i \uparrow n^s_j$ | VNF $n^v_i$ is hosted on substrate node $n^s_j$ |
| $x_{i,j}$ | Boolean variable, true if a virtual function $x_i$ is mapped onto substrate node $j$ |
| $y_i$ | Boolean, true if substrate node is in use |
| $Soft(c,w)$ | Clause $c$ is a soft clause with weight $w$ |
| $route(v^v_i, v^v_{adj}, l^s)$ | True if the adjacent neighbor of $v^v_i$ is $v^v_{adj}$ and it is reached via link $l^s$ |

Section 5.3, an application of the methodology in the context of industrial scenarios is presented with the formal model of the network policies, which includes new properties to verify, such as an isolation and alternative path. In Section 5.4, we present an application of the methodology to the 5G functional split problem, and we discuss the limitations of integer programming in solving the verification problem jointly with VNE, by providing an alternative combinatorial representation of the problem.

## 5.2 Virtual Network Embedding using Substrate Network

A physical topology (also known as substrate network) is described as a weighted undirected graph that we indicate by $G^s = (N^s, L^s, A^s_V, A^s_L)$[79–81]. We demonstrate our methodology with a simple example of a substrate network with two nodes (data centers), three endpoints, and five links, as shown in Figure 5.1. The notation used in our formulation is summarized in Table 5.1 (detailed description can be found in [81]). $N^s$ is used to define the set of physical nodes and $L^s$ is the set of physical links. We model the service graph as another directed graph $G^v = (V^v, L^v, A^v_V, A^v_L)$. $V^v$ is the set of vertices, divided into two disjoint subsets: $N^v$ and $E^v$. $N^v$ is the set of VNFs $n^v$ to be allocated, while $E^v$ is the set of endpoint VNFs $e^v$, whose

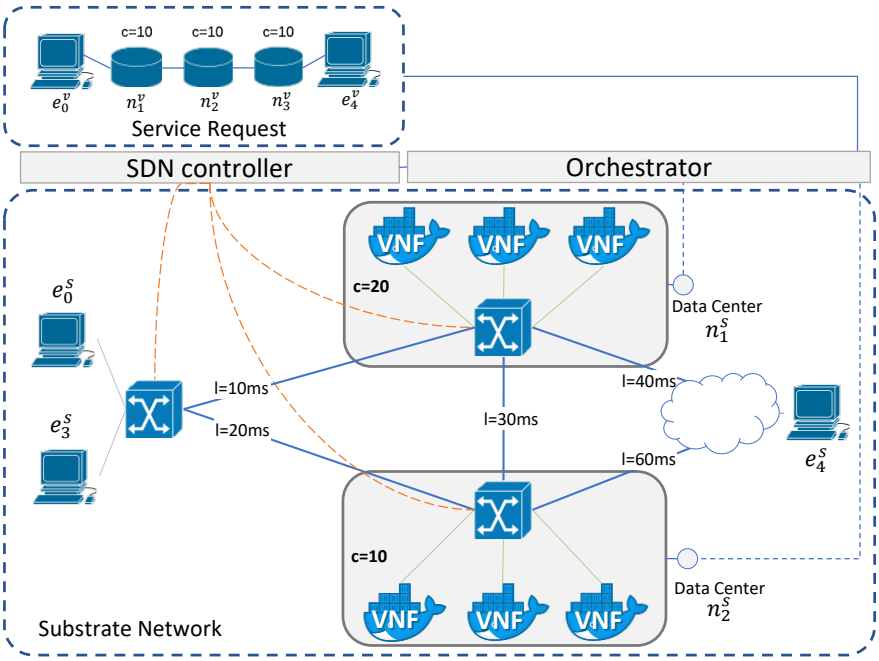


Fig. 5.1 Mapping of the SFC request on top of physical infrastructure

allocation on the substrate network is assumed to be fixed. Notice that the endpoint devices do not necessarily have to be virtualized, and they can co-exist as hardware solutions without affecting our models. $L^v$ is the set of edges (representing the links that connect VNFs with one another). Each edge has a direction, which means that outgoing and incoming packet flows are associated with different edges. Finally, $A_V^v$ and $A_L^v$ are the sets of values that can be taken by the attributes of vertices and edges, respectively. Although this model is general enough, allowing various possible attributes, for simplicity in this work, we consider only a limited set. Each VNF $n^v$ to be allocated has a required storage capacity attribute (i.e., the storage required by the VNF), denoted $storage(n^v)$, a fixed packet processing delay attribute (i.e., time required to process the incoming packets), denoted $lat(n^v)$, and a functional type attribute (i.e., which software program the VNF is running), denoted $func(n^v)$, where $n^v \in N^v$. Depending on the functional type of the VNF, a user may also define the configuration parameters $conf(n^v)$ of that specific VNF, which is another attribute that contributed to defining the functional behavior of each VNF in the service graph. All these VNF attributes, here represented as functions, are provided together with the SG. We stress that the processing delay is inversely proportional to the computational power assigned to a VNF, and in our model it is a property of the VNF and not a demand. For what concerns edges, for simplicity we don't consider

attributes (i.e., $A_L^v$ is assumed to be empty), but our methodology is open to consider these attributes as well.

Finally, we uniquely identify vertices by means of integer indexes, where $v_j^v$ denotes the vertex with index $j$. Depending on the type of vertex, $v_j^v$ may correspond to a service node $n_j^v$ or service endpoint $e_j^v$. For simplicity, we provide an example of a service request forming a VNF chain, rather than a more complex graph (see upper part of Figure 5.1). However, the methodology can be applied to any service graphs, composed of hundreds of VNFs, as it will be clarified in the evaluation section.

The arrangement in the chain is expressed with indexes, i.e., $v_i^v$ is the $i$th VNF in the chain. Figure 5.1 demonstrates the use case of mapping a service function chain consisting of three network functions and two endpoint network devices on a substrate network of two data centers. Each service graph also contains a list of network properties to be verified. Initially, our framework supported two connectivity properties, such as reachability and isolation between endpoints. Later, we introduced other types of properties that are formalized in Section 5.3. Once a service graph $G^v$ with configuration parameters and network properties is fed as an input to the orchestrator, the latter must first provide an assurance that those properties are satisfied in the presence of VNF configuration parameters, and then provide an optimal placement plan onto the infrastructure with Docker.

The position of the endpoint VNFs of the service request $G^v$ is assumed to be already defined in the service request. For this reason, in our scenario, $e_0^v$ and $e_4^v$ are assumed to be mapped onto endpoints $e_0^s$ and $e_4^s$ respectively. In Figure 5.1, $c$ is a shorthand for the storage parameter while $l$ is a shorthand for the latency parameter. The storage constraint is defined as the virtual disk storage required by the VNF VMs, which, in the example, is equal to 10 units for all VNFs. In the figure, the SDN controller is thought of as a part of the Virtual Infrastructure Manager (VIM), which provides centralized control of the data forwarding plane. All data centers in the substrate network are connected to a switch or router and they are managed by the SDN controller. Upon successful verification of the service request forwarding rules, the orchestrator delivers them to the controller. The service request shown in Figure 5.1 is a fairly simple chain of VNFs consisting of a web client ($e_0^v$), firewall ($n_1^v$), a NAT ($n_2^v$), DPI ($n_3^v$), web server($e_4^v$).

The proposed framework for solving the joint optimization and verification problem can be integrated inside an orchestrator, or it can co-exist as an external

component. It receives the service request in addition to configuration parameters and reachability policies. Then these inputs are converted into a set of first-order logic formulas categorized as hard clauses. We present these sets of formulas in Chapter 5.2.1.

Assume that an administrator defines a reachability property between $e_0^v$ and $e_4^v$, where only HTTP packets with allowed payloads must reach the endpoints while the other packets cannot, according to the specifications. It is important to note that we are re-using the same FOL formulas used by Verigraph in order to verify reachability properties. The reachability policy is satisfied if there is any packet flow from $e_0^v$ can reach the $e_4^v$, unless the configuration of the endpoint $e_0^v$ is limited to send a specific type of packet flow. If the endpoint $e_0^v$ is set to send only a certain type of packet flow, reachability property is restricted to these set of packets. Let us assume, configuration of the VNFs in the SG is as follows: the firewall is configured to allow only HTTP packets from $e_0^v$ to $e_4^v$ while the DPI is configured to drop HTTP packets with certain string that is not allowed (i.e., "blacklist"). If the configuration of the endpoint $e_0^v$ is set to send a non-HTTP packet or send a packet containing the "blacklisted" string in the payload, the solver returns an unsatisfiability report to the administrator. This means that with this specific configuration of the endpoint network function, the reachability property is not satisfied, for this specific traffic flow send from the endpoint $e_0^v$. At this point, enforcement of the service is aborted, and a placement plan is not produced. Instead, if the configuration of the endpoint $e_0^v$ is set to send HTTP packets with allowed string (i.e., not in a "blacklist") in the payload or if it is not specified, the satisfiability result with an optimal placement plan is produced. Similarly, an administrator can assert an isolation property between the endpoints (e.g., if affected endpoints need to be isolated). Below we categorize and describe the first-order logic formulas in detail with an example. Similarly, an administrator can assert an isolation property between the endpoints (e.g., if affected endpoints need to be isolated). Below we categorize and describe the first-order logic formulas in detail with an example.

### Resource requirements

In addition to the hard constraints presented by Verigraph, we introduce clauses representing the resource requirements of VNFs. We introduce boolean variables $y_i$ and $x_{ij}$ that take true value when substrate node $n_i^s$ is in use and when network

function $n_i^v$ is hosted on substrate node $n_j^s$, respectively. This last predicate is also denoted $n_i^v \uparrow n_j^s$. The mapping of a service request is then represented by two mapping functions: $M_n$, which maps network functions of the service request onto substrate nodes that meet their resource requirements , and $M_e$, which maps endpoints. $M_n$ can be formally defined as follows. For all $n^v \in N^v$, $M_n(n^v) = n^s$, subject to $n^s \in N^s$, and $n^v \uparrow n^s$, and, for each $j$ such that $n_j^s \in N^s$,

$$\left( \sum_{\forall i | n_i^v \uparrow n_j^s} storage(n_i^v) * x_{ij} \right) \leq storage(n_j^s) * y_j \tag{5.1}$$

where we are assuming the $true$ value of $x_{ij}$ and $y_j$ corresponds to 1, while their $false$ value corresponds to 0. The meaning of these inequalities is that the sum of all storage required by VNFs allocated on a substrate node should be less than or equal to the storage available on that substrate node. A similar formula can be expressed for other attributes as well. In Equation 5.1, we are assuming that network functions from the same service request can share the same substrate node, which is common in NFV systems, e.g., in order to reduce latency. As a concrete example, the service request presented in Figure 5.1 gives origin to the following set of inequalities:

$$10 * x_{11} + 10 * x_{21} + 10 * x_{31} \leq 20 * y_1$$
$$10 * x_{12} + 10 * x_{22} + 10 * x_{32} \leq 10 * y_2$$

Along with these inequalities, we need to express explicitly that each VNF must be embedded to only one node. First, we need to represent explicitly that $M_n$ is a function, i.e. it maps each VNF instance onto exactly one node. For each $i$ such that $n_i^v \in N^v$, this constraint is expressed by the following equation: $\sum_{\forall j | n_j^s \in N^s} x_{ij} = 1$. For our example, such constraints take the following form:

$$x_{11} + x_{12} = 1 \quad x_{21} + x_{22} = 1 \quad x_{31} + x_{32} = 1$$

Finally, for each $j$ such that $n_j^s \in N^s$, in order to correctly bind variable $y_j$ to variables $x_{ij}$, we add the implication $y_j \implies \bigvee_i x_{ij}$, i.e., when a substrate node is selected to be used by the service, we must guarantee that at least one VNF is

deployed on this node. For our example, we express it in the following manner:

$$y_1 \implies x_{11} \vee x_{21} \vee x_{31} \quad y_2 \implies x_{12} \vee x_{22} \vee x_{32}$$

### Routing tables

The forwarding behavior of the service request is expressed by a set of clauses that represent the static routing information of each VNF. Forwarding behavior formulas determine to which VNF to send the packets next. We have presented a detailed discussion of the formulas in our publication [81]. For each VNF $v_i^v$ and next hop $v_{i+1}^v$, we define a predicate $route(v_i^v, v_{i+1}^v, l^s)$ which is true if the next hop of $v_i^v$ is $v_{i+1}^v$ and it is reached via link $l^s$. Each substrate link $l^s$ is associated with the introduced latency $latency(l^s) \in A_L^s$.

Due to the assumption that the placement of an endpoint VNF $e_0^v$ in a chain is known and fixed, the framework constructs a set of soft clauses for each possible substrate node $n_k^s$ onto which $n_1^v$ (the next VNF in the chain) can be allocated:

$$Soft((route(e_0^v, n_1^v, l_{0k}^s) \implies x_{1k}), -latency(l_{0k}^s)) \tag{5.2}$$

where the notation $Soft(c, w)$ specifies that clause $c$ is a soft clause with weight $w$. By defining the opposite of the link latency as the weight, we instruct the MaxSAT solver to minimize the end-to-end delay of the chosen path in the infrastructure.

For instance, the routing table of the endpoint VNF determines the node to which the packet is sent based on the next VNF allocation in the chain. Below we present the actual set of soft constraints of the first endpoint VNF that are generated for the example given in Figure 5.1:

$$\begin{aligned} &Soft((route(e_0^v, n_1^v, l_{01}^s) \implies x_{11}), -10) \\ &Soft((route(e_0^v, n_1^v, l_{02}^s) \implies x_{12}), -20) \end{aligned} \tag{5.3}$$

Instead, for all the intermediate VNFs $n_i^v \in N^v$ in the chain, with $i > 0$, we have a generalized representation:

$$\begin{aligned} Soft((route(n_i^v, n_{i+1}^v, l_{jk}^s) \implies x_{ij} \wedge x_{(i+1)k}), \\ -latency(l_{jk}^s)) \end{aligned}$$

i.e., if an intermediate VNF $i$ sends packets to the next VNF $i+1$ in the chain through link $l_{jk}$, then the corresponding boolean variables $x_{ij}$ and $x_{(i+1)k}$, which denote the position of the VNFs, must be true. If two VNFs are allocated onto the same substrate node, i.e. $j = k$, we have $latency(l^s_{jk}) = 0$, and a soft clause with weight equal to zero is added to the set. For our use case in Figure 5.1 these soft constraints are generated with respect to the input file:

For $n^v_1$:

$$\begin{aligned} Soft((route(n^v_1, n^v_2, l^s_{11}) &\implies x_{11} \wedge x_{21}), 0) \\ Soft((route(n^v_1, n^v_2, l^s_{12}) &\implies x_{11} \wedge x_{22}), -30) \\ Soft((route(n^v_1, n^v_2, l^s_{21}) &\implies x_{12} \wedge x_{21}), -30) \\ Soft((route(n^v_1, n^v_2, l^s_{22}) &\implies x_{12} \wedge x_{22}), 0) \end{aligned} \tag{5.4}$$

For $n^v_2$:

$$\begin{aligned} Soft((route(n^v_2, n^v_3, l^s_{11}) &\implies x_{21} \wedge x_{31}), 0) \\ Soft((route(n^v_2, n^v_3, l^s_{12}) &\implies x_{21} \wedge x_{32}), -30) \\ Soft((route(n^v_2, n^v_3, l^s_{21}) &\implies x_{22} \wedge x_{31}), -30) \\ Soft((route(n^v_2, n^v_3, l^s_{22}) &\implies x_{22} \wedge x_{32}), 0) \end{aligned} \tag{5.5}$$

For $n^v_3$:

$$\begin{aligned} Soft((route(n^v_3, e^v_4, l^s_{14}) &\implies x_{31}), -40) \\ Soft((route(n^v_3, e^v_4, l^s_{24}) &\implies x_{32}), -60) \end{aligned} \tag{5.6}$$

where Equations 5.6 correspond to the routing table of the last endpoint VNF $e^v_4$ in the chain. We do not include the decision variable $x_{4j}$ in this formula, as the placement of this VNF is fixed on a substrate node. Instead, to decide the placement of the VNF $n^v_3$, we generate all possible permutations for $x_{3j}$, where $j \in N^s$. We need to note that these sets of formulas are generated automatically given the ETSI compliant Network Service Descriptor (NSD) file, which consists of parameters following the ETSI MANO specification.

VNE is a multi-objective optimization problem that may involve various cost functions such as minimization of the end-to-end delay, usage of network resources,

maximization of the economical profit (related to embedding cost) and others. Depending on the needs of a network administrator, it is also possible to give priorities to different objectives. In order to do that, the multi-objective functions with lexicographical ordering (i.e., optimization is organized in strict priority levels) can be encoded into MaxSMT using the Boolean Lexicographic Optimization scheme described in [82], by assigning weights to each objective function, where the objectives can be ranked in order of importance. As already noted, weights associated with the soft clauses for the *route* predicates allows us to minimize an end-to-end delay. In this section, we want to introduce the minimization of the number of used substrate nodes, which can be expressed with an additional soft clause generated for each substrate node $n_i^s \in N^s$: $Soft(\neg y_i, L)$, where $L$ is a constant selected according to the target of the minimization: a larger L gives priority to node utilization minimization, whereas a smaller L gives priority to latency minimization. The MaxSMT solver attempts to assign $false$ values to the boolean variables $y_i$ in order to minimize the penalty for falsified clauses in the current model, thus minimizing the number of nodes in use. By introducing the same SG in case of changes in security policies, the solver is able to provide two different outputs. According to the needs of the user, it delivers a new SG deployment on the substrate network or partial solution that will not disrupt the already deployed SG. The latter can be achieved, by setting the variables to a static value, so that the MaxSMT solver will not generate new values to the corresponding variables

Then, by feeding the MaxSMT solver with the conjunction of the clauses expressing the forwarding behavior of the network and the ones representing the placement constraints, we obtain, at the same time, the verification that the specified policies hold, and the optimal placement plan, or an indication that the policies are not satisfied. In the case of our example, the solver produces a satisfiability report with value *true* given to the following variables:

$$x_{12}, x_{21}, x_{31}, y_1, y_2.$$

We can conclude from the output that the firewall VNF $n_1^v$ in the chain is placed on the substrate node $n_2^s$, the NAT $n_2^v$ and the DPI $n_3^v$ on the substrate $n_1^s$. This allocation of the VNFs on the infrastructure introduces a link latency of 90 ms. An experimental assessment of this approach is presented in Section 5.3.2.

Table 5.2 Placement execution time in different substrate networks

| Topology | Nodes | Links | Time (s) |
|---|---|---|---|
| Internet2[83] | 12 | 15 | 0.551 |
| GEANT[83] | 23 | 74 | 17.674 |
| UNIV1[84] | 23 | 43 | 20.684 |
| AS-3679[85] | 79 | 147 | 31.454 |

### 5.2.1 Evaluation and analysis

Different scenarios have been evaluated, each one consisting of a substrate network with a single service request consisting of 4 VNFs to be allocated, with the related reachability property between the endpoints. As it can be seen from Table 5.2, for small topologies the placement and verification algorithm is very fast while for larger scenarios execution time increases. The largest scenario that was tested includes 79 hosts and 147 links, with 4 VNFs to be allocated. In this case, the tool requires an average of 31 seconds.

Algorithms solving the VNE problem come in two forms: offline algorithms and online algorithms. Online algorithms are better suited to deal with high dynamicity, which, however, always comes at the cost of less optimal solutions, relying on heuristics. Moreover, the online VNE problem is more difficult as we need to consider the arrival times of the requests and there are more possibilities of inefficient resource utilization due to time gaps created by earlier mappings. Our version allows tackling the cases where the service requests are issued well ahead of the time when their service will be activated, thus allowing for sufficient time for offline planning. Taking into account that these calculations are performed with an exact method to obtain an optimal solution in offline mode, the computation time is acceptable. As the initial results show promises in small to medium instances, we plan, as future work, to look for improvements of our abstract model in order to cope with even bigger instances and attain further scalability.

The next experiment provides a comparison between the different time taken by the algorithm, considering an increasing number of types of deployment constraints for the VNFs (e.g., the memory that the virtual network function occupies, or the minimum number of cores it needs). These constraints are added for each node, so the total number of hard constraints that are added to the z3 computation is calculated by multiplying the number of types of constraints by the number of nodes that require

a deployment. The constraints have values that do not prevent any of the deployment to be considered (e.g., all nodes require 1GB of RAM while every host has more than enough RAM to satisfy the needs of all the nodes), as this experiment only wants to retrieve the additional computational time needed by z3 to verify these further constraints. If there are not enough resources available for the placement plan, then the solver returns UNSAT without any model, and the time taken to return the result is smaller. The methodology applied for the experiment, as well as the physical topologies used, are the same that were described previously. In the Table 5.3, only the number of constraints is specified because similar performance is obtained regardless of the type of constraints. Moreover, the results show that adding the constraints led to performance degradation, but the main difference is made by the presence or absence of constraints and not by their number. This is due to the internal optimization of the z3 tool respect to the hard constraints.

| Topology | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Internet2 | 2.876 s | 3.519 s (+22.36%) | 3.498 s (+21.63%) | 3.250 s (+13.00%) | 3.628 s (+26.15%) | 3.401 s (+18.25%) |
| GEANT | 5.103 s | 6.257 s (+22.61%) | 6.331 s (+24.06%) | 6.368 s (24.79%) | 6.599 s (29.32%) | 6.919 s (+31.64%) |
| UNIV1 | 10.515 s | 11.855 s (+12.74%) | 11.433 s (+8.7%) | 11.854 s (+12.73%) | 11.228 s (+6.78%) | 11.621 s (+10.52%) |

Table 5.3 Comparison between different number of deployment constraints for the VNFs

## 5.3 Optimize VNF Placement in Industrial Networks

As we enter the Fourth Industrial Revolution, the industrial sector is undergoing fundamental and disruptive changes in the entire product life cycle, focusing heavily on interconnectivity, automation, machine learning, and real-time data. In order to truly realize Industry 4.0, the industrial sector needs to adopt technologies relying on NFV and SDN, as these concepts are becoming increasingly common worldwide. They enable network operators to reduce expenses due to optimized resource utilization and deliver reliable services with respect to various network requirements. One of the fundamental requirements for industrial networks is the end-to-end service delay, because many industrial automation applications have stringent latency requirements to meet the needs of real-time data transmission. Another fundamental requirement is safety, because industrial systems are typically safety-critical, i.e. their failure

can compromize human lives. For the same reasons, considering the higher interconnection and pervasiveness of Industry 4.0 systems, also security is becoming fundamental. Hence, the approach for joint formal verification and optimized VNE presented in general terms in the previous subsection can greatly contribute to achieving all these requirements for Industrial Control Systems (ICS), because formal verification can provide high assurance that safety and security requirements are met while optimized VNE can be used to keep end-to-end delays as low as possible.

Given that the virtual functions can be orchestrated and combined as service function chains between the industrial edge, endpoint devices (Master Terminal Units (MTUs), Remote Terminal Units (RTUs), Programmable Logic Controllers (PLCs), Intelligent Electronic Devices (IED), Human Machine Interfaces (HMI), smart meters, etc.), the resilience delivery problem comes down to select a proper set of network and security functions and to place them across different substrate nodes while meeting ultra-reliable low-latency communications' (URLLC) requirements. On the other hand, the misconfiguration of VNFs discussed earlier can be detected either by straightforward simulation/testing or by formal verification techniques. Since soundness and completeness are not provided by simulation, formal methods are the most suitable solution, also considering that industrial systems are safety-critical. Finally, another way used to improve resiliency is the introduction of fault-tolerance in the network, in the form of alternative paths that can be used in case the main ones fail.

The VNF placement approach presented in subsection 5.4.2 is demonstrated with a use case in this section, while leaving the selection of the network and security functions to be placed in the chain as a separate component to be studied in the future. In addition, we consider the requirement of providing formal assurance that the selected function chain correctly implements the required security and resiliency policies, as presented in the next subsection, and that, at the same time, the latency requirements are met. The formulation of policies introduced here extends the basic connectivity policies introduced earlier and enriches the expressiveness of the formal model. In order to solve the VNE problem in industrial scenarios, our approach accepts similar inputs: (*i*) proper models of VNFs representing both their forwarding behavior and their configuration parameters, (*ii*) a model of the substrate network, and (*iii*) the resiliency policies that must hold in the industrial network. Given these inputs, it generates a formally verified placement plan.

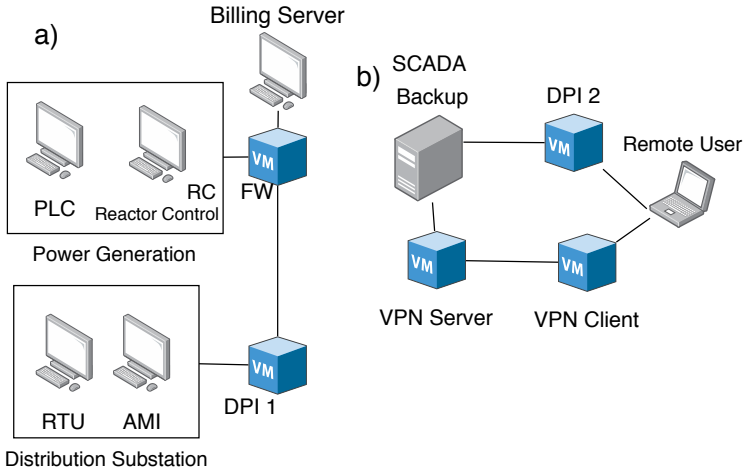


Fig. 5.2 Illustration of different scenarios a) reachability and isolation b) alternative path

### 5.3.1 Policy model

We consider the connectivity policy rule $p$ as a tuple $p = (t, s, d)$, where $s$ and $d$ represent the source and the destination of a communication, respectively, and $t$ is the *type* of policy described below.

Source and destination $s$ and $d$ are both subsets of SG vertices ($s, d \subseteq V^v$). They can include either single VNFs or zones (e.g., whole IP subnets). For what concerns the type $t$, in this thesis we consider it can take one of three different possible values, but our approach is flexible enough to accommodate other types. In particular, we consider *reachability (R)*, *isolation (I)*, and the presence of *alternative (A)* paths.There is reachability from network node/zone $s$ to $d$ if there exists at least one path that connects $s$ to $d$, and in this path, there are no VNFs that block this communication. While there is isolation from $s$ to $d$ if there is no path that connects $s$ to $d$, or there are paths, but in each one of them, there is at least one VNF that blocks this communication. Finally, there are alternative paths from $s$ to $d$ if there are at least two disjoint paths that connect $s$ to $d$, and that do not include VNFs that block this communication. By disjoint paths, we mean paths that have no shared edges and vertices.

Two examples are illustrated in Figure 5.2, which represents two possible service graphs that can be deployed in the infrastructure. In the first scenario, depicted in Figure 5.2(a), the service graph implements an industry recommended network security guideline, where the Power Generation field industrial control systems must

be isolated from a Distribution Substation. This is done by means of a firewall. The graph also includes an advanced metering infrastructure (AMI) linked to a back-end server, for instance, a Billing Server and, to increase the overall security of the service, a DPI module. In this case, the service request may include two connectivity policies to be verified: clearly, the isolation between the Power Generation field and the Distribution Substations, but also the reachability between the Billing Server and the AMI device. A reachability between the Billing Server and the AMI device defined in scenario (a) can be expressed as:

$$p_{a_1} = (R, \{Billing\ Server\}, \{AMI\})$$

whereas the isolation rule that isolates two zones can be expressed as:

$$p_{a_2} = (I, \{PLC, RC\}, \{RTU, AMI\})$$

In the second scenario, shown in Figure 5.2(b), the service graph represents a reliable SCADA backup system where the administrator requires the SCADA backup endpoint to be reachable by means of two redundant paths with different security features. Those are provided by a normal connection that passes through a DPI function in the former and by a VPN tunnel in the latter. The administrator can, therefore, be interested in verifying that the service request as given satisfies an alternative path property, i.e. that, with the deployed graph and configurations, the target is actually reachable by means of the two paths. We may express the policy on path alternative required in this scenario in the following way:

$$p_b = (A, \{Remote\ User\}, \{SCADA\ backup\})$$

In both cases, and in general in the industrial field, latency minimization or bounding is key to guarantee proper operation and performance, while taking care of data center utilization is important for several reasons ranging from cost reduction to power saving. Hence, in this use case, we give priority to the minimization of the latency between the endpoints. We should emphasize that, the introduction of this policy formalism will not affect the existing formulation consisting of FOL formulas presented in Section 5.4.2. Instead, it formalizes the representation of complex network properties that complements the original reachability property adopted from Verigraph. For instance, *alternative (A)* paths policy is constructed by asserting the reachability property on two different chains in an SG.

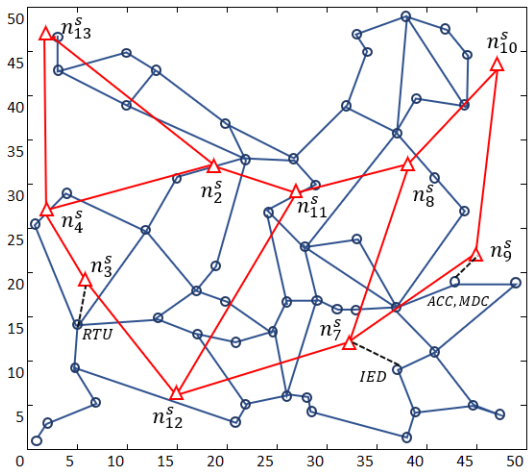


Fig. 5.3 Experimental Industrial facility [1] (redrawn). △ a substrate network of 10 nodes and ◯ IEEE 57 bus test system

### 5.3.2 Evaluation and analysis

In order to evaluate our model in an industrial scenario, we reuse the same topology considered for the substrate network of DCs and Smart Grid devices presented in [1], where the Internet2 network topology forms the substrate network and the IEEE BUS 57 test system topology hosts the Smart Grid endpoints. The experimental topologies are depicted in Figure 5.3. We exploit the same network topology generator, GENSEN [86], to generate a realistic geographical distribution of grid edge endpoints. Similarly, the link latency is assumed to be directly proportional to the Euclidean distances between the endpoints and substrate nodes.

The IEEE Bus 57 Test Case of the American Electric Power System in the Midwestern, US, includes several sections representing information from different devices in the power grid. Buses are nodes in the network or substation locations, and branches are the connections between buses. Each power system bus works as a gateway router that is connected to the closest substrate network node either through wired or wireless links (e.g., xDSL, LTE). The gateway routers aggregate traffics from endpoint VNFs to be forwarded to other endpoints or substrate nodes throughout the substrate network. In the evaluation phase of the tool, we select random nodes from the test system topology and map the endpoint VNFs of the service on them. We illustrate the physical connections that link Smart Grid nodes to substrate nodes with dashed lines, as shown in Figure 5.3.

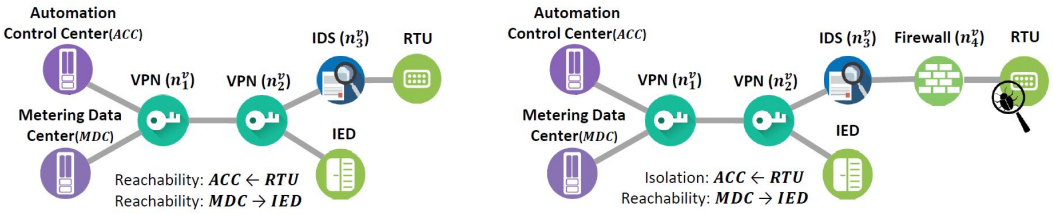


Fig. 5.4 Examples of Service Graph

We consider an initial scenario where the service graph (represented in Figure 5.4(a)) consists of SCADA commodity devices to perform various grid control applications, SCADA slaves that interact with the control devices, and security functions in between. For this example, endpoints correspond to the Automation Control Center (ACC), Metering Data Center (MDC), Remote Terminal Unit (RTU), and Intelligent Electronic Device (IED) in the Smart Grid network. These endpoints are connected through an encrypted channel with the help of two VPN termination network functions $n_1^v$ and $n_2^v$. Additionally, there is an IDS network function $n_3^v$ between the endpoints ACC, MDC, and RTU.

In our service model, we assume the location of the endpoints to be fixed and associated with specific substrate nodes (i.e., they are not considered in the placement procedure), whereas the network functions need to be placed in the substrate network. Concerning the verification aspects, we define two reachability policies in this SG from RTU to ACC and from MDC to IED.

Let us assume now that a compromised RTU is detected in the field network. As a reaction, the NFV orchestrator proposes an updated service graph depicted in Figure 5.4(b) to mitigate the impact of cyber-attacks. This updated SG includes a firewall VNF $n_4^v$ to block the packets generated by the RTU device from proceeding to the control center, while the reachability requirement between ACC and RTU is converted to an isolation policy.

Here we present the results of the updated graph (b), which has different placement in the substrate network compared to the original graph (a). The high-level representation of the updated service that includes both aspects related to the service graph (i.e., which network functions implement the service, how they are interconnected among each other) is fed as an input to our tool along with the configurations of these network functions. The list of these configurations is given below:

- VPN access $n_1^v$ is configured to have the IP address of the VPN exit $n_2^v$ gateway as a single parameter.
- VPN exit $n_2^v$ is configured to have the IP address of the VPN access $n_1^v$ gateway as a single parameter.
- IDS $n_3^v$ contains an entry in its "blacklist" table with not allowed function code equal to $\{43\}$. Any packet containing a function code equal to 43 expected to be blocked.
- Firewall $n_4^v$ contains an entry {src:$RTU$ dst:$ACC$ sport:$*$ dport:$*$ proto:$*$} in its ACL table, which blocks packets sent from $RTU$ to $ACC$.
- ACC is set to generate a packet with a function code different from $\{43\}$. In this way the packet originated from ACC will not be blocked by the intermediate VNFs.

The required storage capacities of each VNF are integers with a uniform distribution between 10 and 50, whereas the available storage capacity of each substrate node varies between 100 and 150. An overall processing delay of each VNF is randomly determined by a uniform distribution between 50-100 [87]. As the configuration parameters of the involved VNFs satisfy the new isolation property in Figure 5.4 (b), a latency-aware optimal placement of VNFs in substrate network is generated as an output:

$$x_{1,9}, x_{2,7}, x_{3,12}, x_{4,3}, y_3, y_7, y_9, y_{12}$$

In this list, we highlighted only the variables whose value is true, which shows the allocation of the VNFs and occupancy of the substrate nodes. From this output, we can conclude that the VPN termination $n_1^v$ is placed on the substrate node $n_9^s$, the VPN termination $n_2^v$ on $n_7^s$, the IDS $n_3^v$ on $n_{12}^s$ and the firewall VNF $n_4^v$ on the substrate node $n_3^s$.

Using this exemplified use case, we investigate the performance results comparing the computation time of the proposed approach to the APPLE[49] for various data sets. We feed our tool with this updated SG, including 8 VNFs on real data sets of Table 5.4. For each substrate network topology, we use the same IEEE BUS 57 test system topology for Smart Grid endpoints. We limit our study to this data set only, for the Smart Grid, but any other data set in the "IEEE Common Data" format

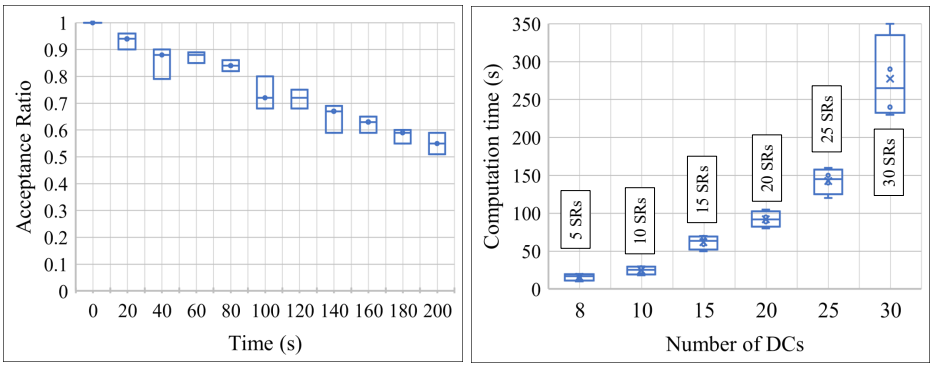


Fig. 5.5 (a) Acceptance ratio over time (b) Computation time depending on the arrival of service requests (SRs) and size of the substrate network.

is applicable to our approach. The demonstrated results reflect the computation time of the latency minimization problem under various conditions. Each of these scenarios consists of a substrate network and service request to be allocated. All evaluations are executed on a workstation with 32GB RAM and an Intel i7-6700 CPU.

It is evident from the table how the average computation time of the proposed approach for the Internet2 topology adopted from [1] is low and certainly compliant with the industrial requirements (less than 1 sec for 10 nodes and 13 links). However, GEANT, UNIV1, and AS-3679 network topologies show significant computation overhead due to the much larger network sizes considered. It is important to note that the different combinations of configurations of network functions and the number of properties to verify have a small impact on the complexity of the problem in contrast to the size of the topology.

This approach allows to input multiple service requests arriving at the same and obtain the placement plan for each request simultaneously. However, this approach is costly in terms of computation time, as the number of constraints increase depending on the number of requests. To solve this issue, multiple requests can be solved sequentially or even in parallel, depending on the availability of support for parallelism by the MaxSMT solver. In our case, we can exploit parallel disjoint tactics provided by z3 MaxSMT solver that try different strategies in solving the problem. However, this analysis is out of the scope of this paper and remains to be investigated in a separate study.

Table 5.4 Comparison between Verification and Optimization computation times

| **Topology** | **Nodes** | **Links** | **Time (O+V)** | **Time (O)**[49] |
|---|---|---|---|---|
| Internet2[83] | 10 | 13 | 0.6 | 0.029 |
| GEANT[83] | 22 | 36 | 15.4 | 0.1 |
| UNIV1[84] | 23 | 43 | 22.2 | 0.235 |
| AS-3679[85] | 79 | 147 | 35.1 | 3.013 |

Instead, the widely accepted sequential approach is more promising in terms of scalability and covers realistic scenarios, where service requests arrive over time. Fig. 5.5 (a) depicts the acceptance ratio (average number of requests accepted for hosting in the physical substrate) of the algorithm over time. As shown here, acceptance ratio declines with the increasing service request arrival rate, as the solver is not able to find valid assignments within the substrate network. Fig. 5.5 (b) shows the computation time required to embed service requests with an arrival rate of 5 requests over time. In order to keep the acceptance ratio of the algorithm equal to 1, we increase the size of the substrate network over the x-axis. This intensive simulations provides an estimate on the performance of the tool. However, the network reconfiguration of the Smart Grid is performed (not frequently) due to many reasons such as weather factors, protection malfunction, service upgrade or cyber-attacks. The service instantiation time of major NFV Orchestrators in case of a reconfiguration is usually in the order of minutes. In comparison, that the computation time introduced by our framework is satisfactory under these circumstances.

Compared to the time of optimization (O) only given in [49], the time taken by our optimization and verification tool is higher. However, it is important to mention that the placement optimization and verification *(O+V)* of an SG is a problem fundamentally different and more complex than the simple VNE problem. APPLE[49] (Automatic aPProach for poLicy Enforcement) is an optimization engine that solves the VNE problem in a short time, but specific aspects of NFV such as forwarding behavior, chaining, and models of network functions are not addressed by the authors of the tool. These additional details that must be considered in our work certainly introduce further time complexity. However, as part of possible future work, it would be possible to develop heuristic-based algorithms to profitably consider much larger network sizes.

## 5.4 Multi-Objective Functional decomposition in 5G

### 5.4.1 Background

We also would like to provide a brief introduction to the 5G Radio Access Network (RAN) functional decomposition problem, which can be easily modeled as an instance of the VNE problem. Starting from the 3G, the demand for data services, that is, "mobile Internet access", has suddenly emerged, gradually replacing voice services and becoming the primary source of revenue for operators. There are many types of data services, but because of limited network resources, it is impossible to guarantee that all services can be carried out at full speed. Therefore, different data services need to be prioritized. In order to better adapt to a variety of application scenarios and customer needs, and to serve the digital transformation of various industries, the communications industry has been actively developing 5G [88] technology, an improved version of 4G network through cloudization and virtualization. Network slicing is one of the enablers of 5G networks to address the isolation of the traffic of various industries. By splitting the virtual network into slices on the substrate network, it isolates the traffic coming from wide range of fields such as autonomous driving, smart grid, telemedicine and industrial control.

Network slicing is the way of decoupling a hardware infrastructure into multiple virtual end-to-end chains. Each slice may have independent resources, and they are completely isolated. Any issue involving a single slice will not impact others. Combined with 5G application scenarios[24], operators can define different slices according to different service types, meet different users' requirements for various delays, throughput, capacity, and efficiency, and can give users a better experience. With this respect, NFV and SDN technologies provide essential technical foundations for slicing. The operator selects the virtual and physical resources it needs for the specified communication service type in accordance to the SLA (Service Level Agreement). It includes multiple parameters such as the number of users, QoS, and bandwidth, and different SLAs that will define different types of communication services.

According to different business requirements, the wireless network side protocol stack functions can be flexibly customized and divided too. Compared with the two-level structure of the baseband processing unit (BBU) and radio remote unit (RRU) of the 4G wireless access network (RAN), gNB of the new 5G air interface

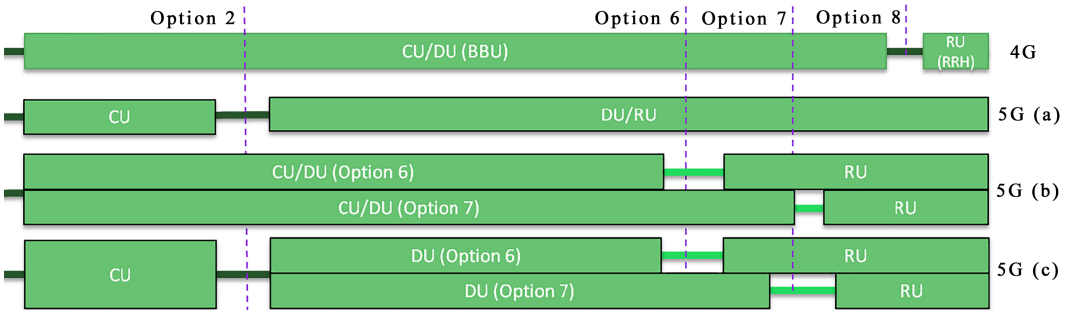


Fig. 5.6 Split options in the 4G and 5G signal processing chain

can adopt a three-level structure of a centralized unit (CU), a distribution unit (DU), and a radio unit (RU) [24]. Figure 5.6 shows this difference highlighting the two-level structure (4G line), where CU an DU are unified and isolated from RU. The section of the traditional BBU responsible for the asynchronous processing will be split and redefined as CU, responsible for processing non-real-time protocols and services, mainly including packet data convergence protocol (PDCP) and radio resource control (RRC); part of the BBU and the traditional RRU Merged into RU, which mainly includes the underlying physical layer (PHY-L) and radio frequency (RF); the rest of the BBU functions are renamed as DUs, and includes radio link control (RLC), media access control (MAC) and high-level physical layer (PHY-H) functions. There are multiple deployment methods for CU and DU of 5G RAN, three of which shown in Figure 5.6. The simplistic scenario given as 5G (a), where RU and DU are integrated, can be achieved if CU can connect using fiber. When CU and DU are combined (i.e., Figure 5.6 5G (b)), the structure of 5G RAN is similar to that of 4G RAN, and it is a two-stage structure of fronthaul and backhaul, but the interface rate and type of 5G base station (gNB) have changed significantly. When CU and DU are split (i.e., Figure 5.6 5G (c)), it will evolve into three levels of fronthaul, midhaul, and backhaul. In the 5G initial stage, RAN network deployment will be dominated by macro stations; with 5G large-scale commercial use, macro stations and room-based base stations will be deployed in different scenarios. The specific deployment method is divided into distributed wireless access networks (DRAN) and Centralized Radio Access Network (CRAN). The cloudification of the 5G access network will promote large-scale CRAN deployment with separated CU, DU, and RU. The decision to choose and optimally allocate the right split in the infrastructure (e.g., three-tiered substrate network shown in Figure 5.7) will highly depend on the services, and this introduces a new problem to solve: optimal functional split.

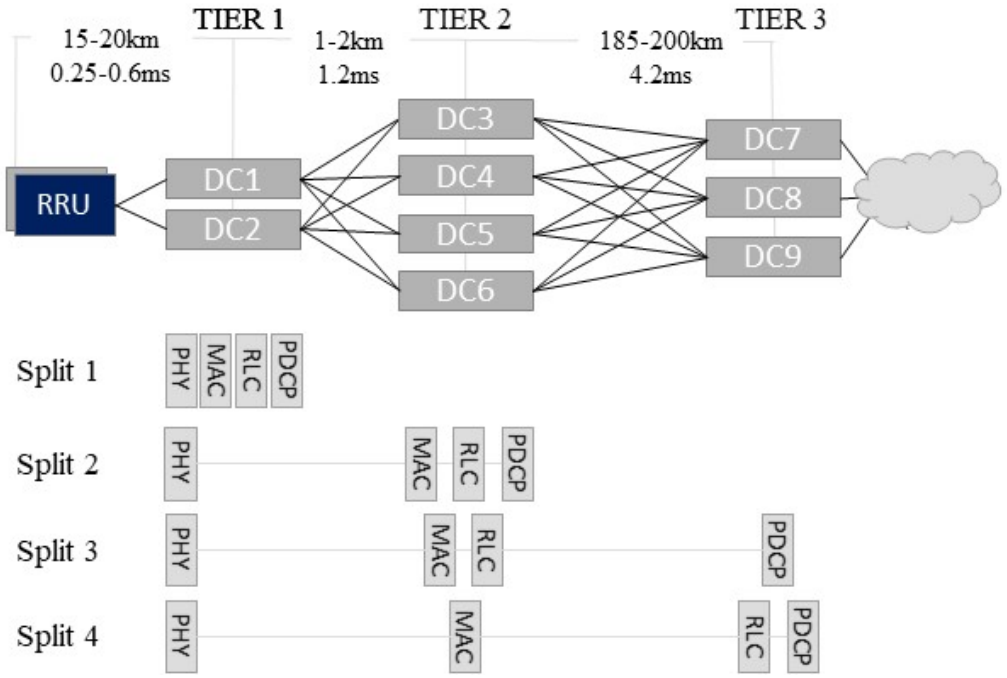


Fig. 5.7 3 tier RAN functional split in view of different deployment scenarios

In the next section, we show that the joint verification and optimization approach also allows us to determine the optimal functional split and provide an end to end-reachability guarantees between the endpoints.

The existing literature formulates the VNE problem using Integer Programming (IP) formulation, which is limited to a set of constraints over binary, integer, or real variables. Instead, the approach presented in this thesis allows us to model the problem and using very expressive constraints. These constraints include configuration parameters of VNFs, forwarding behavior of the service graph, and complex security policies, in addition to the usual constraints of the VNE problem. The presence of these additional constraints allows us to jointly perform VNE and formal verification of network properties. To show this and to compare the performance of two approaches, the Mixed-Integer Quadratically-Constrained Programming (MIQCP) model is proposed in this section. Continuing the discussion on uRLLC, eMBB, and mMTC, we analyze how low-latency and high bandwidth requirements of these traffic classes are met by providing different split options in 5G RAN. The analysis is performed by finding the optimal placement for the service function chains based on optimization goals for different network slices. First, we formulate the placement problem as MIQCP for the RAN requirements in terms of latency and throughput.

Then we adapt our approach presented in Section 5.4.2 to provide formal verification and optimal placement of VNFs with propositional logic formulas in Conjunctive Normal Form.

### 5.4.2 Mixed Integer Quadratically Constrained Programming

In this section, we list the required input considered in the placement algorithm, and then we present the MIQCP formulation of the function placement problem.

Following the same approach that we outlined for the general approach, we model the substrate network, where RAN network function chains are placed, as a connected directed graph, $G = (V, L)$. $V$ is the set of vertices, divided into two disjoint subsets: $N$, $E$ (as shown in Section ). The data rate available is $d(v, v')$ for every edge $(v, v') \in L$ and the network links are directed edges with latency $l(v, v')$. Upon the arrival of a service request, an orchestrator component of NFV must decide how to optimally allocate the different ordered sets of radio network functions onto the substrate network nodes. In these requests, depending on the user-plane (UP) and control-plane (CP), different orders of functions are specified, which define flows between fixed start (e.g., user equipment) and end (e.g., IP services) points. With the set $E_{pairs} \subseteq E \times E$, we define pairs of start and end point nodes belonging to different flows.

Our goal is to analyze how low-latency and high bandwidth requirements of uRLLC, eMBB,and mMTC traffic classes are met by providing different split options in 5G RAN. The analysis is performed by finding the optimal placement for the service function chains based on optimization goals for different network slices. The problem of finding the optimal functional split is analogous to the VNE problem, which is the main reason why we want to compare the MIQCP model to a MaxSMT model introduced earlier. We model the placement optimization problem as a MIQCP with respect to a number of used network nodes, latency, and data rate. The capacity of network nodes and requirements of different network functions then characterize the input. The notation used in our derivation is summarized in Table 5.5, where $l(u, u')$ is used to define latency between $u$ and $u'$ and $rem_{v,v'}$ is used to represent a remaining data rate on $(v, v')$, when services utilize the same link.

Table 5.5 Summary of key notations

| Domain | Parameter | Description |
|---|---|---|
| $\forall v \in V$ | $c(v)$ | Substrate node computational resources in $v$ |
| $\forall (v,\ v') \in L$ | $l(v,v')$ | Latency of $(v,v')$ |
| | $d(v,v')$ | Data rate capacity on $(v,v')$ |
| $\forall (u,u') \in U_{pairs}$ | $d_{req}(u,u')$ | Data rate demand of $(u,u')$ |
| $\forall (u,u') \in U_{pairs}$ | $l(u,u')$ | Latency between $u$ and $u'$ |
| $\forall u \in U$ | $p(u)$ | Substrate node demand of $u$ |
| $\forall (a,a') \in E_{pairs}$ | $paths(a,a')$ | Paths between $a$ and $a'$ |
| | $l_{req}(a,a')$ | Required latency between $a$ and $a'$ |
| $\forall u \in U, \forall v \in V$ | $m_{u,v}$ | $u$ mapped to $v$ |
| $\forall (v,v') \in L$, $\forall x,x' \in V$, $\forall (u,u') \in U_{pairs}$ | $e_{v,v',x,y,u,u'}$ | $(v,v')$ belongs to path between $x$ and $y$, where $u$ and $u'$ are mapped to |
| $\forall v \in V$ | $used_v$ | At least one request mapped to $v$ |
| $\forall (v,v') \in L$ | $rem_{v,v'}$ | Remaining data rate on $(v,v')$, when services utilize the same link |

**Placement Constraints**

By Formula $\forall u \in U: \quad \sum_{v \in V} m_{u,v} = 1$, all virtual nodes must be mapped onto a single substrate node, iff. the request is to be embedded. If at least one function is mapped on a substrate node, we denote it as "used" with the constraint: $\forall u \in U, \forall v \in V: m_{u,v} \leq used_v$

Resource requirements such as a required storage of all functions mapped to a node should be less than or equal to available resources in that node:

$$\forall v \in V: \quad \sum_{u \in U} m_{u,v} \cdot p(u) \leq c(v) \tag{5.7}$$

**Path Related Constraints:**

If $(u, u')$ pairs are mapped to the $(x, y)$ nodes and an edge in the request belongs to a path between nodes $v$ and $v'$, then the path is created between those network nodes:

$$\begin{aligned} \forall(v,\ v') \in L, \forall x,\ y \in V, \forall(u,\ u') \in U_{pairs} : \\ M \leq m_{u,x} \cdot m_{u',y} \end{aligned} \tag{5.8}$$

For the simplicity, we use variable $M$ in place of $e_{v,v',x,y,u,u'}$.

Moreover, each functional split has different latency requirements for data transfers across the function locations; involves different amounts and types of resources (computing power, link capacities); and brings different cost savings and performance benefit. This is expressed by the following constraints:

$$\begin{aligned} \forall(v, v') \in L, x, y \in V, (u, u') \in paths(a, a') : \\ M \cdot l(v, v') \leq l_{req}(v, v') \end{aligned} \tag{5.9}$$

According to uRLLC requirements where the sum of latencies of all edges of a flow should be less than the maximum latency given for that flow is expressed as follows:

$$\begin{aligned} \forall(a, a') \in E_{pairs} : \\ \sum_{\substack{(v,v') \in L, x, y \in V, \\ (u,u') \in paths(a,a')}} M \cdot l(v, v') \leq l_{req}(a, a') \end{aligned} \tag{5.10}$$

The total bandwidth consumed by all the requested source-destination traffic flows going through an edge should not exceed the bandwidth capacity of this edge:

$$\begin{aligned} \forall(v, v') \in L : \\ \sum_{(u,u') \in U_{pairs}, \forall x, y \in V} e_{v,v',x,y,u,u'} \cdot d_{req}(u, u') \leq d(v, v') \end{aligned} \tag{5.11}$$

Sum of latencies of network edges belonging to a path where $u$ and $u'$ are mapped gives the end-to-end latency of this path:

$$\forall (u,u') \in U_{pairs}: \quad l(u,u') = \sum_{x,y \in V,(v,v') \in E} M \cdot l(v,v') \tag{5.12}$$

Finally, the remaining data rate of an edge is calculated as,

$$\forall (v,v') \in L: \quad rem_{v,v'} = d(v,v') - \sum_{\substack{(u,u') \in U_{pairs}, \\ \forall x,y \in V}} M \cdot d_{req}(u,u')$$

### Objectives

Combinations of different objectives can also be achieved with all the variations of IP. Combinations for different use case scenarios can result in a different mapping of the network functions into the network graph. This section describes the multi-objective algorithm that aims to find the best candidate node for embedding each VNF of a chain. This algorithm consists of three steps:

*Minimizing the number of utilized nodes in the network:*

$$minimize \sum_{v \in V} cost(used_v) \tag{5.13}$$

This objective aims to minimize the cost of utilized DCs and applicable for all types of services. The cost helps businesses to rent the hardware or software from infrastructure providers, paying only for what they use. However, it might concentrate on the placement of functions, which causes congestion in the network. In fact, centralizing RAN functions within a cloud infrastructure significantly improves cost, but they can only be centralized so far as the latency budget is still met, plus other factors such as transport capability.

*Maximizing the remaining data rate on network links:*

$$maximize \sum_{(v,v') \in E, v \neq v'} rem_{v,v'} \tag{5.14}$$

In order to avoid the congestion in the network, we introduce this objective that maximizes the data rate on the links leaving more bandwidth for future requests, and it is only applied for eMBB services.

*Minimizing the latency of the created paths:* As previously stated, 5G is supposed to provide users with unprecedented experience with ultra-low latency. However, there might be multiple paths available between the endpoints that are all compliant with low latency constraints. In these instances, selection of the path with the minimum latency is favorable and it can be expressed with the following objective:

$$minimize \sum_{(a,a')\in l_{req}} \Big( \sum_{P\in paths(a,a')} \Big( \sum_{(u,u')\in P} l(u,u') \Big) \Big) \tag{5.15}$$

It is important to note that we were able to transform most of the propositional calculus statements presented in this thesis into MIQCP. However, the large number of generated variables used in this approach caused memory failures for bigger instances. Moreover, MIQCP this encoding was unfeasible to describe the forwarding behavior and configuration parameters of network functions. This validates our choice of formulating the joint verification and placement problem as MaxSMT model, which has a higher descriptive power. In the next section, we present the performance comparison of these two approaches.

### 5.4.3 Evaluation and analysis

For the evaluation of the MaxSAT model in the 5G scenario, we have performed the placement for different sets of deployment requests using the objectives defined. The costs obtained by the solvers with MaxSAT and MIQCP formulations, in the case of uRLLC are identical, even if the placement plans were different for some of the instances. This is explained by the equivalence of different placement plans.

Evaluation results are given in Figure 5.8 showing the scalability of the proposed approach with respect to the simple RAN service request (i.e., Figure 5.7) that includes PHY, MAC, RLC, and PDCP. The network functions are modeled as radio functions without configuration parameters. They act as a simple forwarding function. The results from our experiments show that the computational cost for providing formal assurance about reachability in addition to optimal embedding of virtual functions is greater than the case of the MIQCP model. However, we

should emphasize that the MIQCP solution only provides an optimal placement plan and does not provide formal verification of the network properties in presence of VNF configurations. We conclude that the MaxSMT model is usually slower than the MIQCP model in finding optimal solutions, but MaxSMT solution allows us to obtain optimal placement plan with a formal assurance that all network policies are hold in presence of configuration parameters of complex network devices. In this chapter, we showed that the closest combinatorial encoding of the same MaxSMT problem cannot provide the same level of complexity in defining the complex network constraints.

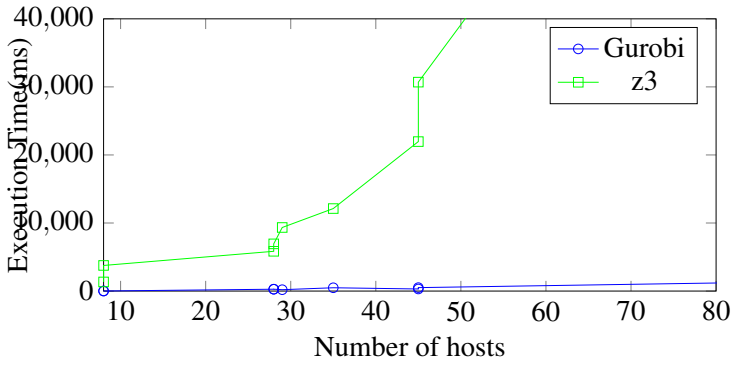


Fig. 5.8 Copmarison between MaxSAT and MIQCP execution time

# Chapter 6

# VErified REFinement and Optimized Orchestration

## 6.1 Requirements and challenges

Misconfiguration of *Network Security Functions (NSFs)* such as firewalls and VPN terminators have recently become the third most critical exploit for cybersecurity attacks, as Verizon underlined in its most recent Data Breach Investigations Report[89]. This problem is intrinsic of the manual approach by means of which network administrators work since typically filtering or protection rules are distributed on the NSFs with heuristic and suboptimal criteria based on human common sense [90].

This critical risk motivates the introduction of *automated policy-based network security management tools*: they can assist human beings in the creation and configuration of a security service by means of an automatic process in charge of creating the policy of each NSF so as to respect some security requirements, also called intents, expressing the goals to be compliant with. The advantages of pursuing *Security Automation* are evident: some examples are avoidance of human errors, automatic conflict analysis of the policies, and formal verification of their effective correctness. A fundamental benefit is, nonetheless, the possibility to pursue *optimality*: automated algorithms can, in fact, reach the optimal outcomes more easily than manually. This has undoubtedly a critical impact on the resources that are needed to create an effective security service. However, despite all these positive prospects, the research is still moving its first steps towards a fully automated and

optimized approach; altogether, the number of developed tools is still limited in this context [61]. Moreover, as demanded by any automatic process, agility is a fundamental requirement to support a similar approach. From this point of view, novel networking technologies such as *NFV* [91] and *SDN* [92] can bring a heavy contribution.

A problem that nevertheless arises in this context is the huge variety of tools that can be used to orchestrate the virtual functions. Although the *ETSI* defined a standard architecture [23], each NFV and cloud orchestrator has different peculiarities and characteristics [93], which reflect the purposes they have been built for and the targets of their developers or vendors. This has a severe impact on the portability of any automatic process which would be in charge of the allocation and configuration of virtual security functions, because it should adapt itself to work with a huge variety of different APIs, input, and output data formats.

Given all these considerations, in this section, we propose an extension of the Verifoo framework presented in Chapter 5 and we call it VErified REFinement and Optimized Orchestration (Verefoo). Its purposes are to refine high-level security requirements, which are expressed with a human-friendly language, into the optimal allocation scheme and configuration of the NSFs on a *SG* representing the network service [94]. This step is performed in a *correctness-by-construction* fashion, so that there is formal assurance of its correctness; moreover, the computed results are optimal with regard to a set of cost functions, such as minimization of the number of installed functions or number of rules in their configurations. The architecture of Verefoo is general enough and can be used with several types of VNFs. Currently, it has been developed only for firewalls. We plan to address the consideration of other NSFs in future work.

Finally, the framework is designed to be able to deploy the virtual functions and configure them through direct interaction with some orchestrators, without any manual operation. In particular, the two well-known orchestrators that have been integrated with Verefoo at the moment are Open Baton, which traditionally manages Virtual Machines in an NFV environment, and Kubernetes, which can be also in charge of the orchestration of Dockers in a cloud scenario. The remainder of this chapter is organized as follows. In Section 3.3, related work is presented. In Section 6.2, the proposed framework and its main implementation components (e.g., allocation, selection and placement) are given, whereas Section 6.3 details the model,

which will formulate both the automatic configuation and the mapping problem in FOL. After presenting the various integrations of the framework with orchestrators in Section 6.4, Section 6.5 details the performance evaluation results.

## 6.2 The Verefoo framework

In this section, we will illustrate the main principles and purposes of the approach we followed in designing the architecture of Verefoo; in particular, we will provide a complete overview on the framework, describing the tasks that each module is in charge of and explaining how it has been integrated with the most well-known NFV and cloud orchestrators.

### 6.2.1 The Verefoo approach

Verefoo manages the creation, configuration and orchestration of a complete end-to-end network security service following a modular approach, that is reflected by the design of the framework itself.

First of all, Verefoo automatically performs, on a provided *Service Graph*, an optimized allocation and configuration of the *Network Security Functions (NSFs)* that are necessary to fulfill an input set of *Network Security Requirements (NSRs)*, which can be expressed by the service designer – i.e. the person in charge of creating a network service – by exploiting a high-level language. High flexibility is granted by allowing the users to define the NSRs repository and to create the catalog of functions available in the system.

The input Service Graph is made by network functions without any security capability; so, initially it must be automatically transformed into a logical topology, called *Allocation Graph*, where, between any pair of virtual functions, an Allocation Place is created. Each Allocation Place is a placeholder position where a NSF could be allocated, if it is needed in order to satisfy the input security constraints. An example is showed in Figure 6.1, where it is worth underlining that each element, from the NAT to the load balancer, are actually VNFs instead of physical appliances. Then, to establish the optimal allocation scheme of the NSFs and their configuration, a *correctness-by-construction* approach is followed, by the definition of a *MaxSMT*

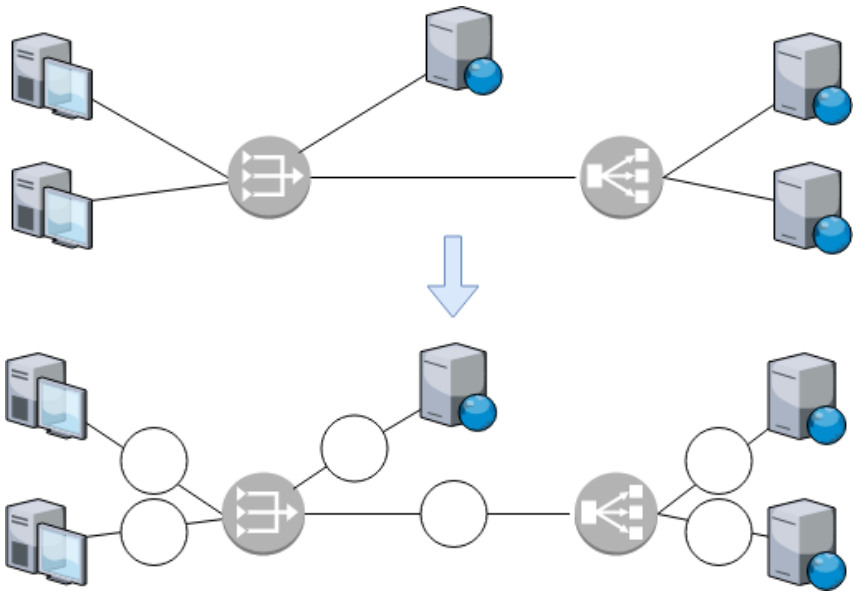

Fig. 6.1 Example network consisting of possible allocation options.

problem that is in charge of automatically choosing which network functions are needed and where to allocate them; thus, also formal assurance of the correctness is provided. Optimality is another key factor that has been considered to pursue these objectives: actually, the best solution is the scenario where the minimum number of VNFs for security functions are introduced and the minimum number of rules are configured in their policies.

This first step is completely performed on a logical level. Then, after the creation of this virtual security service, a second objective is to establish the optimal placement of each function on the physical servers that compose the substrate network. Other cost functions are considered in this phase, such as minimization of the latency between VNFs or resource consumption. Besides, since each NSF is characterized by configuration rules that are expressed with a medium-level language that provides abstraction from implementation, a translation is needed to get the vendor-specific configuration of each virtual function.

Finally, the service is set up by means of an integration with cloud orchestrators, in order to provide security properties for the communication between end points or networks. It is important to remark that this result is achieved just starting from a Service Graph and a set of security requirements as inputs, thanks to the fully

automatic algorithms that are exploited in the overall approach, as it will be more extensively explained in the next subsection.

### 6.2.2 Framework overview

Figure 6.2 presents a complete overview of the framework, so that we can provide a brief description of each component to give a general idea of the workflow.

According to our vision, the user of the NFV orchestrator - i.e. the service designer - is able to introduce as input:

- a set of Network Security Requirements (NSRs) to express the security constraints which must be fulfilled, by exploiting a high-level or a medium-level language [61] depending on the experience level of the user, through a Policy GUI which makes the creation of the requirements easier;
- a Service Graph (SG) or, in alternative, directly an Allocation Graph (AG) through a Service GUI, which provides access to a Network Functions Catalog (NF Catalog) from which the user can decide which functions – simple network functions or also NSFs – immediately allocate on the graph.

A preliminary phase is represented by the *Policy ANalysis (PAN)*; the goal of this module, which receives the NSRs as input, is to perform a conflict analysis exploiting well-known techniques [95, 11, 96, 97], to establish if some of the requirements are in conflict, and to create the minimal set of constraints which must be respected in the network. It can provide an early non-enforceability report to the service designer in case the input security requirements are characterized by mistakes which cannot be solved by means of this automatic process but require a reformulation by the user.

If the specified security requirements are expressed in a high-level language, the *High-to-Medium (H2M)* module performs a refinement to get a corresponding set of medium-level NSRs, which contain all the useful information for the future creation of the policies of the NSFs automatically allocated on the virtual graph and the low-level configuration of the VNFs placed on the substrate network. These security requirements are based on isolation and reachability-based network properties.

Then, a key role is covered by the *NF Selection (SE)* module; based on the input high-level and medium-level NSRs, it decides which NSFs are required to satisfy

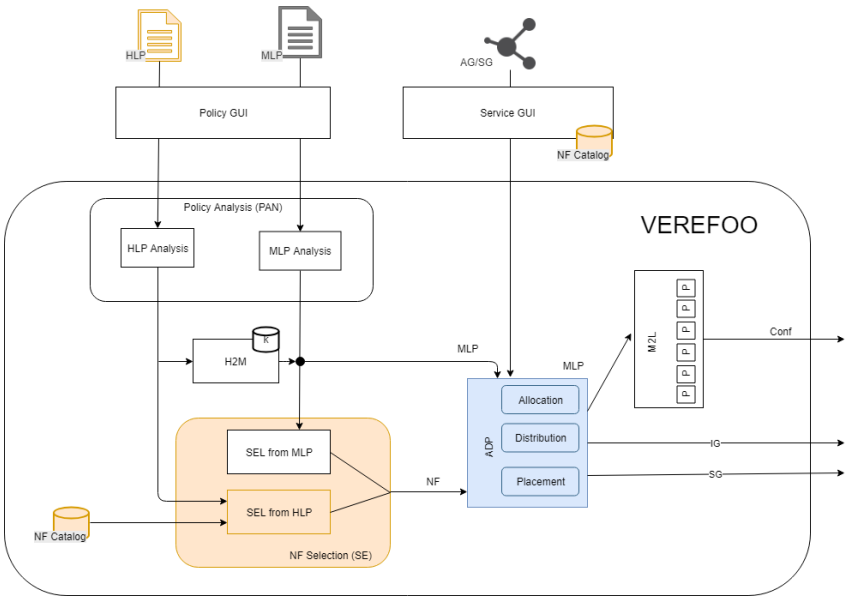


Fig. 6.2 Verefoo general architecture.

them, choosing them from a pre-built catalog, that is the same list the service designer has access through the Service GUI. This step requires an optimization process by means of which the optimal set of NSFs is selected, even though this operation does not exploit any knowledge about the topology of the Allocation Graph. This result is achieved in the following way: at first, SE receives, from a module outside the framework, the list of the instances of the required security capabilities and then it searches, among the functions present in the NF Catalog, which ones support the requested capabilities and selects the optimal functions, taking into account available physical resources, so as to be able to allocate them in the physical servers. The selection of functions is subject to the conditions imposed: in the first place the functions must be able to support the capabilities, but it is also necessary that certain physical resources are available in order to be able to place the functions on the servers. In addition, the choice is subject to optimizations: it is possible to reduce the cost of the functions as well as reduce the amount of resources (e.g. RAM) needed.

The *Allocation, Distribution and Placement (ADP)* module is one of the main elements of the architecture, whose purpose is to compute a Service Graph with the added NSFs and to decide the VNFs placement on the substrate network receiving as input the medium-level NSRs, the list of selected NSFs and the original Service Graph or directly the Allocation Graph. The ADP module uses *z3Opt* [98] as a MaxSMT solver and Verefoo to provide three main features:

- given a list of NSFs selected by the NF Selection module, it orchestrates their allocation on the Allocation Graph — received in input or obtained from the processed Service Graph -– in order to satisfy the input NSRs expressed by means of the medium-level language;
- in contemporary with the allocation phase, a second task is the distribution of the policy rules on the allocated NSFs, always expressed in medium-level language but not necessarily identical to the input NSRs formulation, because the policy rules can be minimized according to optimization goals;
- in a secondary step, after the creation of the Service Graph enriched with the NSFs, the VNFs implementing the network functions of the original Service Graph and the added NSFs are placed in the physical infrastructure following the principle of minimizing the resource consumption and at the same time the medium-level policy rules of the NSFs are translated into the low-level configuration of the VNFs themselves.

An additional output of the ADP element is the list of medium-level policy rules by means of which each network function instance must be configured; then, the corresponding low-level configuration that depends on the specific implementation of the deployed function is generated by the *Medium-to-Low (M2L)* module, which performs a translation of the vendor-independent expressions into the rules which must be set on the proper function.

### 6.2.3 Integration with NFV and Cloud Orchestrators

The security service that has been created by the ADP module of Verefoo must be then instantiated by creating the virtual processes on the physical servers. For this purpose, communication with orchestrators is needed.

To reach this objective, the complete architecture of the solution we are proposing in this thesis does not include only Verefoo, that represents the logical computing core, but also other tools that work as intermediate interfaces between the orchestrators and Verefoo itself. Figure 6.3 shows four examples of interfaces that can be developed and included in our framework:

- *VeriBaton* can provide integration with Open Baton;

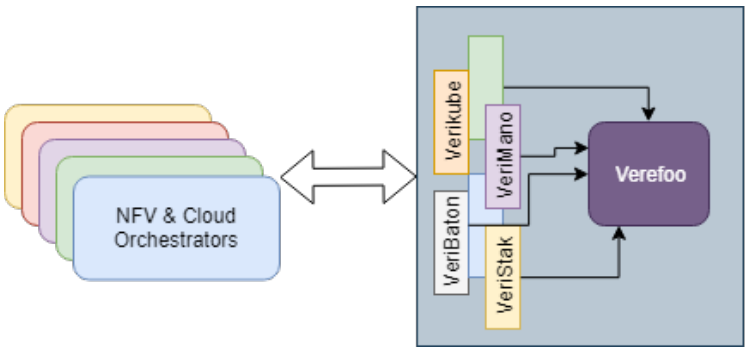


Fig. 6.3 Integration architecture.

- *VeriKube* can provide integration with Kubernetes;
- *VeriMano* can provide integration with OSM;
- *VeriStak* can provide integration with OpenStack Tacker.

Each interface offers RESTful APIs to interact with both the ADP module of Verefoo, from which it receives all the needed information to create and configure the security service in the virtualized network, and the NFV or cloud orchestrator, that is in charge of the set-up and management of the life-cycle for each VNF. The presence of these interfaces is transparent to the user, who interacts exclusively with Verefoo as beforehand described, but their role is fundamental. In fact, if all the current orchestrators such as those that have been named are not able to automatically create and configure a network service, this becomes possible by means of the integration with our framework, that is thus achieved without requiring to exploit Verefoo and the orchestrator separately.

To make clearer how this integration is achieved, in the following section we first provide the formal model of the presented technique and provide the details of the integration with two interfaces, VeriBaton and VeriKube, alongside with the validation of the proposed approach through various use cases in Open Baton and Kubernetes.

## 6.3 The Model

An important extension added to Verifoo is the possibility to have VNFs without any configuration or with partial configuration, giving to the tool itself the task of

providing the missing configurations as an output. Verifoo generates the configuration with the objective of satisfying all the requested policies while minimizing the number of generated rules in order to achieve it. For instance, for a firewall, this translates into the generation of the rules that decide if a received packet needs to be dropped or not. The auto-configuration does not affect the deployment in any way, since all the VNFs that are declared in the service graph will be deployed onto a host in the physical topology even if they still have an empty configuration after the z3 computation (this would mean that even without adding any rules, the policies are satisfied).

In addition to the FOL formulas presented in Chapter 5.4.2, we introduce additional soft clauses, which allow the solver to decide the possible values for the variables that are not initialized. In this chapter, we have a number of non-limiting assumptions on the Firewall VNF model. Throughout the thesis we also refer to firewalls as packet filters that filter on the basis of the IP 5-tuple. A firewall has a default action and that all the specific rules are of the same type, which is opposite to the default value (i.e. all ALLOW or all DENY). In this way, we ensure the absence of conflicts. Moreover, we assume that that a maximum number of rules equal to the number of security policies expressed by the user, as we are sure that no more than that number is necessary.

Internally, Verefoo describes the Firewall VNFs that require the autoconfiguration differently from the models described in Chapter 4. Below we provide the additional constraints introduced to the existing Firewall VNF model, which allow obtaining configuration parameters that satisfy the predefined network policies. Clauses listed in Equations 6.1 allow the solver to decide values for each field of the IP packet (i.e., $src, dst, protocol, src\_portanddst\_pott$). It uses a set of soft clauses that can be falsified that are associated with the ACL of a Firewall, which, initially, must not have any entry in the ACL table if the Firewall VNF is not configured. For this reason, we initialize these values to a "null" (or a value that has the same meaning for the specific context, e.g. zero for a TCP port), so that the output values of these values are produced as "null" or 0. During the translation phase of the generated output model, Verefoo excludes all variables having a "null" value, which is equivalent to having no rules on this specific Firewall. In conjunction with these soft clauses, also new hard constraints are declared to model the behaviour of the specific VNF. The general idea is that following the VNF model, z3 decides which soft clauses will have values different from their default ones in order to satisfy the requested policies.

Not using the default value will introduce a penalty that z3 tries to minimize, hence resultant numbers of significant configuration parameters are the least possible. In particular, for a firewall, the rules that are generated express an action (ALLOW or DENY) that is always the opposite of the default one in order to avoid any conflicts.

The soft clauses that are declared for a firewall for each rule are the following:

$$\begin{aligned} &Soft(src == null, k, \text{“}rules\text{“}) \\ &Soft(dst == null, k, \text{“}rules\text{“}) \\ &Soft(protocol == 0, k, \text{“}rules\text{“}) \\ &Soft(src\_port == null, k, \text{“}rules\text{“}) \\ &Soft(dst\_port == null, k, \text{“}rules\text{“}) \end{aligned} \tag{6.1}$$

where the first argument of the $Soft$ function represents the constraint that can be falsified, $k$ is a constant that defines its weight, and the third argument is a label assigned to differentiate between the classes of soft clauses (the weights of the constraints in each class are optimized independently from one another). The previously shown soft clauses are associated each with one of the fields that compose a firewall rule. In (6.1), $protocol$ is an integer while $src$, $dst$, $src\_port$ and $dst\_port$ are modeled as belonging to type DatatypeSort. This type is provided by z3 to allow the developers to define more complex data structures. In fact, $src$ and $dst$ are abstractions of IP addresses and, as such, they are made of four different integers that compose the address DatatypeSort. For instance, a source field of the packet is defined as $p0.src\_i$. Moreover, also the port fields are declared as a particular DatatypeSort in order to consider the possibility to have an interval (e.g. 10-80) and not only a single value. This latter DatatypeSort is therefore composed of two integers that represent the start and the end of the interval. To instruct z3 on which values are possible for the mentioned DatatypeSort, new hard constraints have been added.

For the IPs:

$$\begin{aligned}
&\forall\{n0, n1, p0\} : \\
&\quad recv(n0, n1, p0) \implies \\
&\qquad p0.src._0 > 0 \wedge p0.src._0 < 255 \wedge \\
&\qquad p0.src._1 > 0 \wedge p0.src._1 < 255 \wedge \\
&\qquad p0.src._2 > 0 \wedge p0.src._2 < 255 \wedge \\
&\qquad p0.src._3 > 0 \wedge p0.src._3 < 255 \wedge \\
&\qquad p0.dst._0 > 0 \wedge p0.dst._0 < 255 \wedge \\
&\qquad p0.dst._1 > 0 \wedge p0.dst._1 < 255 \wedge \\
&\qquad p0.dst._2 > 0 \wedge p0.dst._2 < 255 \wedge \\
&\qquad p0.dst._3 > 0 \wedge p0.dst._3 < 255
\end{aligned} \tag{6.2}$$

The formula (6.2) ensures that every packet that is exchanged between the VNFs has correct IPs. This is because for every packet that is received, there is one that is sent and vice versa, if it doesn't break any other constraints (e.g. the packet has some blacklisted fields). Therefore, it would be redundant to repeat the same constraint also for the send function since if there is a send, then there is also a corresponding receive with the same arguments. The only exceptions for this assumption are the endpoints, for which a packet can be sent even without a corresponding receive (client endpoint), or a packet can be received even without a corresponding send (server endpoint). For the ports intervals:

$$\begin{aligned}
&\forall\{n0, n1, p0\} : \\
&\quad recv(n0, n1, p0) \implies \\
&\qquad p0.src_port.start > 0 \wedge p0.src_port.end < MAX_PORT \\
&\qquad \wedge\, p0.dst_port.start > 0 \wedge p0.dst_port.end < MAX_PORT
\end{aligned} \tag{6.3}$$

where MAX_PORT is defined as the constant 65535.
For the IP related fields, the assignment in (6.1) can be therefore considered as a view at a higher level of the following formulas (which are the ones that effectively

are fed into z3):

$$\begin{aligned} &Soft(src_0 == 0, k, \text{“}rules\text{“}) \\ &Soft(src_1 == 0, k, \text{“}rules\text{“}) \\ &Soft(src_2 == 0, k, \text{“}rules\text{“}) \\ &Soft(src_3 == 0, k, \text{“}rules\text{“}) \end{aligned} \tag{6.4}$$

with $src = (src_0, src_1, src_2, src_3)$. To keep this consistency, one hard constraint is added to correlate the *src* variable, that will be used to check for matching fields in the packets as shown afterwards, with the $src_i$, to which a value will be assigned by z3.

To improve the potential of the autoconfiguration task, each byte of the IP addresses can also be assigned to be equal to a wildcard in order to allow z3 to generate rules that exploit that feature. This obviously leads to a possible further minimization of the total number of rules. To express this possibility in z3, the assumption that has been made is that the value "-1" is considered to be the wildcard. This allows the solver to prefer wildcards when using them is possible. Therefore, to implement this feature the following declarations must be added:

$$\begin{aligned} &Soft(src_0 == -1, c, \text{“}wildcards\text{“}) \\ &Soft(src_1 == -1, c, \text{“}wildcards\text{“}) \\ &Soft(src_2 == -1, c, \text{“}wildcards\text{“}) \\ &Soft(src_3 == -1, c, \text{“}wildcards\text{“}) \end{aligned} \tag{6.5}$$

In (6.5) it is important to notice that the class is different from the previous declaration (the weight $c$ can also be different from the previous $k$ but it doesn't matter as they are in different classes). Using a different class makes z3 capable of distinguishing between a “null” rule and a rule with wildcards otherwise it will use them indistinctly. Here only the src variable is shown but deriving the variables for the dst one is straightforward.

A generic rule is then declared as a boolean condition that returns true if the fields of a generic packet p0 match the soft clauses declared in (6.1). At a high level, the rule

declaration can be written as:

$$\begin{aligned} rule = (&p0.src == src \wedge \\ &p0.dst == dst \wedge \\ &p0.protocol == protocol \wedge \\ &p0.src\_port == src\_port \wedge \\ &p0.dst\_port == dst\_port) \end{aligned} \tag{6.6}$$

where the equalities $p0.scr == src$ and $p0.dest == dst$, also consider the possibility to have wildcards (e.g. the comparison between 10.0.0.1 and 10.0.0.-1 returns true). This is achieved by transforming the mentioned equalities as follows:

$$\bigwedge_{\forall i \in \{0,1,2,3\}} p0.src_i == src_i \vee src_i == -1 \tag{6.7}$$

The condition declared in (6.6) defines the way a packet matches a firewall rule. For each requested policy, one rule is created to work as a placeholder considering the worst-case scenario (i.e. every policy needs one rule to be satisfied), however z3 will always assign significant values only to the least number of rules. Nevertheless, every firewall in the service graph has a number of placeholder rules equal to the set of policies even if a particular firewall is never traversed by some of the flows declared in those policies (i.e. the firewall is in neither of the paths that link a source to a destination specified in a policy, thus it will never be able to affect that specific flow) leading to the declaration of some unnecessary variables. A brief discussion about the performance is carried out in the next section.
At this point the firewall behaviour is modelled with a set of hard constraints that use the rules declared in (6.6). The model may take different form based on the default firewall action that has been declared in the input service request. If no default action is declared by the user in the input, in order to have a more conservative approach, the default action is set to DENY, i.e. drop every packet. If the default action is ALLOW, the firewall is then modelled by means of the following formulas:

$$\begin{aligned} &\forall \{next, p0\} : \\ &\quad send(firewall, next, p0) \implies \\ &\quad\quad \exists \{previous\} | recv(previous, firewall, p0) \wedge \neg rule \end{aligned} \tag{6.8}$$

$$\forall\{previous, p0\}:$$
$$recv(previous, firewall, p0) \wedge \neg rule \implies \exists\{next\}|send(firewall, next, p0) \tag{6.9}$$

Formula (6.8) states that if a firewall can send a packet p0 to a next node *next*, it is necessary that the same packet can be received by the same firewall from a previous node (*previous*) and the rule condition, which established if a packet has to be discarded by the firewall, returns false (i.e., p0 does not match the rule). Formula (6.9), instead, expresses the inverse implication. In the general case where there are multiple rules configured in a firewall, the *rule* condition can be expressed as:

$$\bigvee_{\forall i|r_i \in R} r_i \tag{6.10}$$

where $R$ is the set of all the placeholder rules of the firewall. This ensures that the correct behaviour is applied even if there are multiple firewall rules to consider.

If the default action is DENY the constraints are the same as before but there is no negation in front of the *rule* condition as shown below:

$$\forall\{next, p0\}:$$
$$send(firewall, next, p0) \implies \exists\{previous\}|recv(previous, firewall, p0) \wedge rule \tag{6.11}$$

$$\forall\{previous, p0\}:$$
$$recv(previous, firewall, p0) \wedge rule \implies \exists\{next\}|send(firewall, next, p0) \tag{6.12}$$

In (6.11) and (6.12) the logic is inverted with respect to the one described previously, in order to fit the opposite default action.
When enforcing the requested policies, one last condition is added to enforce the

constraints of the specified flow. The constraint for each policy is the following:

$$\begin{aligned} \forall \{next, p0\} : \\ & send(firewall, next, p0) \implies \\ & \quad p0.lv4proto == specified\_lv4proto \\ & \quad \wedge\, p0.src\_port == specified\_scr\_port \\ & \quad \wedge\, p0.dst\_port == specified\_dst\_port \end{aligned} \tag{6.13}$$

where the $specified_*$ variables are provided by the user. Having the possibility to falsify the assignments in (6.1), z3 can choose the right value for a soft clause in order to satisfy the requested policies in accordance with the default action of the firewalls, using as few rules as possible.

## 6.4 Implementation

In order to meet functional requirements, different design approaches of the Verefoo framework have been considered, taking into account various use case scenarios and used service orchestrator capabilities and characteristics. Service orchestrators are a relatively new technology, yet different products are already available from open-source communities (e.g., Juju, Open Baton, Kubernetes, MAESTRO). The implementation of the Verefoo use cases will integrate the framework in existing orchestration frameworks, such as Open Baton and Kubernetes.

### 6.4.1 Open Baton Verefoo integration

A contribution to Open Baton project has been the most flexible solution considered, as deep integration with the MANO architecture. The main goal of this integration is to extend the capabilities of the Open Baton orchestrator in terms of Service Graph verification and validation, by integrating the tool Verefoo. This solution allows to interact with the orchestrator introducing graph validation and optimization in the phase of Network Service catalog upload, thus allowing onboarding of a Service Graph only after a validation check, rejecting formally invalid descriptors, and possibly updating the graph to provide optimal placement plan. To achieve this, we meet the following principles:

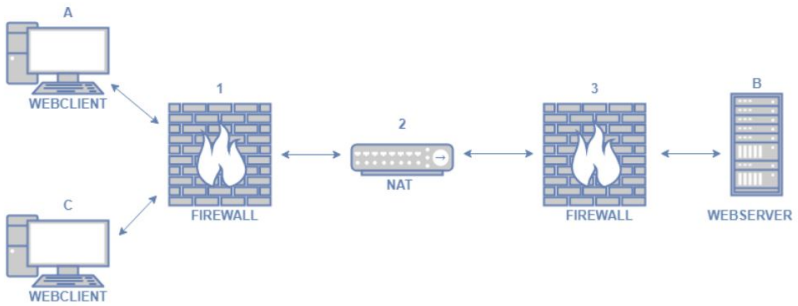


Fig. 6.4 Service request with optional nodes.

- *Interface compatibility*: the interface used to interact with Verefoo service should match completely the interface exposed by Open Baton for NSD onboarding. In this way, the end user can be unaware of Verefoo presence if not interested, and behave as it was interacting with Open Baton; network services can be developed following ETSI data model, making Verefoo validation capabilities pluggable at the user discretion. Verefoo becomes this way a sort of "proxy" which could be used depending on the needs, with NSD instances built directly for Open Baton. As communication with Open Baton happens through a REST interface over HTTP, Verefoo will be itself a RESTful API server.

- *Input validation*: Verefoo acts as a validator component for the input provided to Open Baton, implying that an invalid Service Graph will be blocked before reaching Open Baton with suitable feedback for the user. It is in charge of verifying that nodes are correctly organized in a chain, and policies specified as input such as reachability and isolation between VNFs are satisfiable, assuring that a service present in the catalog once deployed does behave as expected.

- *Graph optimization*: once received the optimal service configuration from Verefoo, the original input should be modified according to it. Possible scenarios include removal of nodes from the graph and automatic configuration firewalls, which should be reflected on the NSD to be uploaded to Open Baton.

To verify the capabilities of the framework, we designed an NSD instance representing a common use case of the tool and fed it as input to Verefoo. Figure 6.4 describes visually the configuration of the test instance including two web

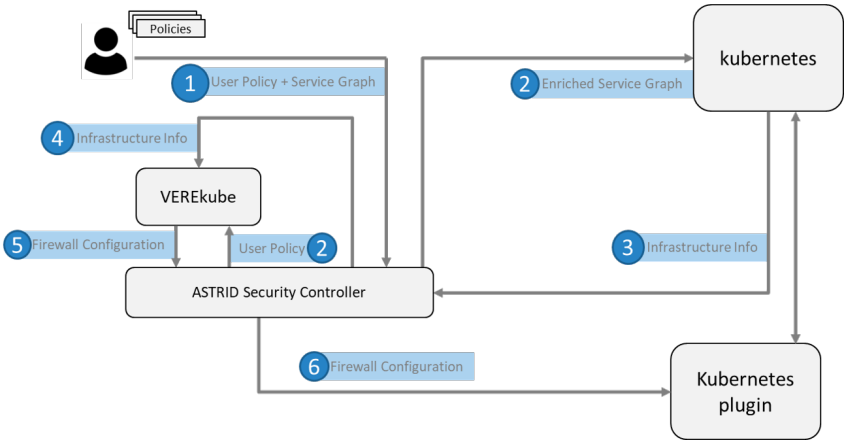


Fig. 6.5 General architecture of the ASTRID framework for Kubernetes integration.

client nodes, two firewalls and one NAT VNFs connecting to a web server. The requirements of the service request are:

- reachability is required between node A and node B;
- isolation is required between node C and node B;
- node 1, 2 and 3 are optional and are not configured.

Once Open Baton (ETSI) compliant NSD description of the service is provided as an input to the orchestrator, we expect to minimize the number of NSFs, while satisfying these user requirements. As a result, we obtain a report that the service validation is successful. Moreover, the Service Graph has been updated as expected, where node 3 has been removed, while node 1 has been configured to allow traffic directed from node A to node B, denying everything else.

This validates the proposed approach, successfully presenting a solution capable of translating the information model based on the ETSI specifications to the application-specific format defined by the verification engine albeit the substantial differences in data representation and structure between Verefoo and Open Baton orchestrator.

### 6.4.2 Kubernetes Verefoo integration

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications. Kubernetes enables to quickly deploy

containerized applications, scaling it according to the user needs, without having to stop anything in the process. It is made to be portable, extensive and self-healing, granting an easier management from people who have to administrate the system. The Kubernetes orchestrator is the second orchestrator used for demonstration and validation. We integrate the Verefoo framework with Kubernetes to provide lightweight monitoring and enforcement hooks in each virtual function, which can be dynamically programmed, to react to management events and security alerts, by invoking specific security services. In this integration Verefoo takes as input the service topology, the current network configurations, and the security policies, and returns as output the configuration of the security hooks. In the current implementation, the scope is limited to automatic firewall configuration. Figure 6.5 presents the general ASTRID (AddreSsing ThReats for virtualIseD services) project workflow of the Verefoo integration with Kubernetes orchestrator. ASTRID is a European project[99], whose goal is to isolate the detection and analysis tasks from the service graph, by delegating this task to a central orchestrator unit. This allows service providers to make transparent and shift security, privacy and responsibilities of third parties. Our implementation involves a number of components of ASTRID framework, and it starts with user delivering policies and Service Graph to the controller, at this time the enriched Service Graph with all the information model required by Kubernetes is defined and delivered. At this point security controller sends the user policy to Verefoo. Kubernetes provides the infrastructure information based on the deployed graph to the controller, which is then delivered to Verefoo. In the next step, Verefoo computes formally verified configuration parameters of the firewalls, in order to satisfy the user policies and delivers them to the controller, which then sends it back to context broker. Context broker is in charge of enforcing the firewall rules in the NSFs.

In this context, the results of some performance tests carried out on the introduced integration are illustrated, in order to show which goals have been achieved and to understand which limitations should be refined in the future. We focus on two metrics – numbers of Allocation Places and of Network Security Requirements –, to perform the scalability tests by increasing one metric to an higher value, while keeping the others fixed, to understand to which extend the first metric is scalable. Given this assumption, the results of the performance tests which have been carried out to understand the scalability of the developed framework are showed in Figure 6.6 for the Allocation Places and for the Network Security Requirements. They have

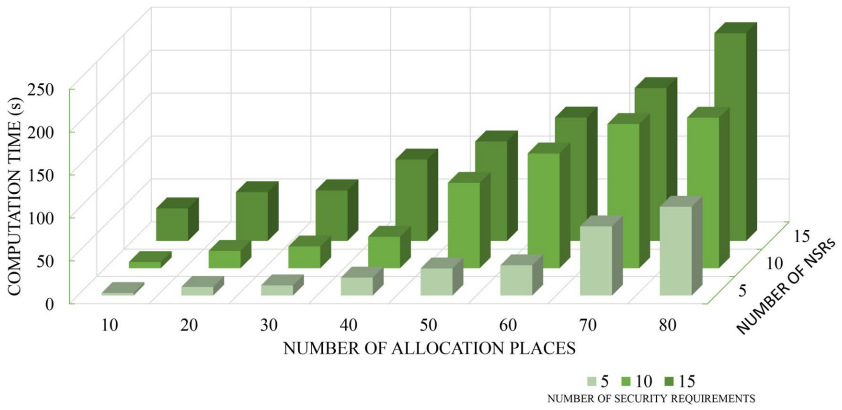


Fig. 6.6 Results of scalability tests.

been achieved with a machine characterized by a 3.40 GHz Intel i7-6700 CPU and 32GB of RAM.

This chart shows that the computation time does not increase exponentially either with the number of Allocation Places or with the number of NSRs. This result is particularly positive, given the intrinsic worst-case computational cost of a MaxSMT problem, that belongs to the NP-complete class. Consequently, the framework is able to manage Service Graphs of medium-big dimensions, with a high number of links and end points, providing the optimal solution to the presented problem, if the number of Network Security Requirements is not extremely high. These results clearly show that the network structure and the number of security requirements strongly influence time performance, but provides us a strong hint about the fact the the approach we are following is feasible and worth to be further explored.

## 6.5 Performance validation results

This section discusses the autoconfiguration performance with a particular focus on the scalability issues that currently affect it. In order to evaluate the performance of the proposed approach, a series of charts will be shown, coming from our experiments. All evaluations are executed on a workstation with 32GB RAM and an Intel i7-6700 CPU. In these charts, the plotted lines represent the temporal trend of the autoconfiguration task when the set of policies grows (the number of firewalls is fixed to one). To have an idea of which feature causes the greatest slowdown, the

measurements have been performed considering various types of autoconfiguration, which are:

- Basic autoconfiguration, where the tool is restricted to generate rules that are composed only by IPs, without the possibility of using the wildcards (in this case the addresses are modelled in z3 using an EnumSort instead of a DatatypeSort)
- Quintuple autoconfiguration, which adds the possibility to generate rules that include also the protocol and the source and destination ports (the IPs are still EnumSort).
- Wildcards autoconfiguration, where the addresses are considered as a more complex data structure (here the DatatypeSort is used) and the firewall can use the wildcards to further minimize the number of rules. However, in this configuration only the IPs are present in the generated rule, therefore there are no protocol nor ports.
- All features autoconfiguration, in which the generated rules contain IPs, protocol and ports, with the IPs that can also have wildcards.

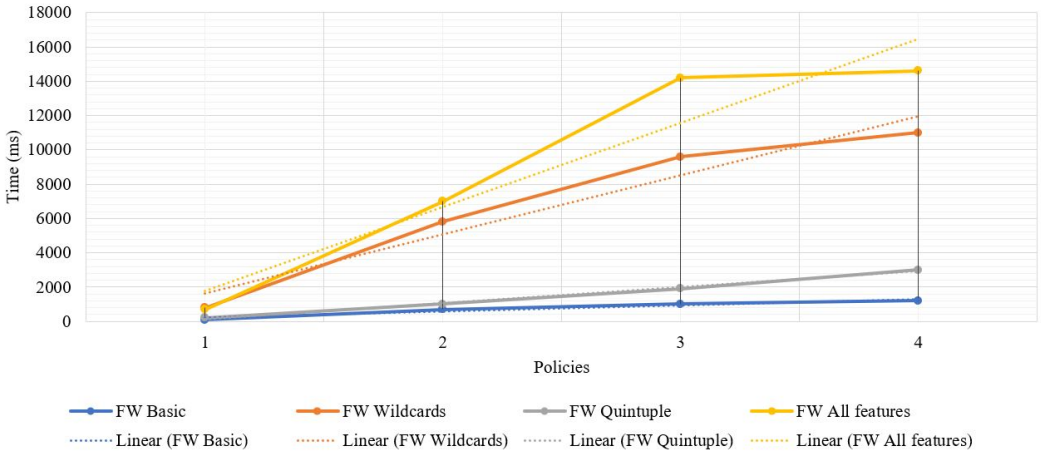


Fig. 6.7 Refinement comparison with a single VNF

As shown in chart 6.7, using a DatatypeSort to model the IPs (wildcard autoconfiguration), causes a significant slowdown with respect to its counterpart with a simple EnumSort (basic autoconfiguration). This concept can be further analyzed in the next chart. In that chart, both the basic and the quintuple autoconfigurations

are the same as in the chart above, however, in addition, two other lines have been plotted. One of them refers to the temporal trend of the autoconfiguration task for a DPI, whose model is exactly the same as the firewall one, but the packet field that the DPI checks is set to be the body of the packet (modelled as an integer). The other added line is the result of a firewall that checks only the protocol field, which is also an integer field. This last configuration has been evaluated only for testing purposes and has obviously no real application.

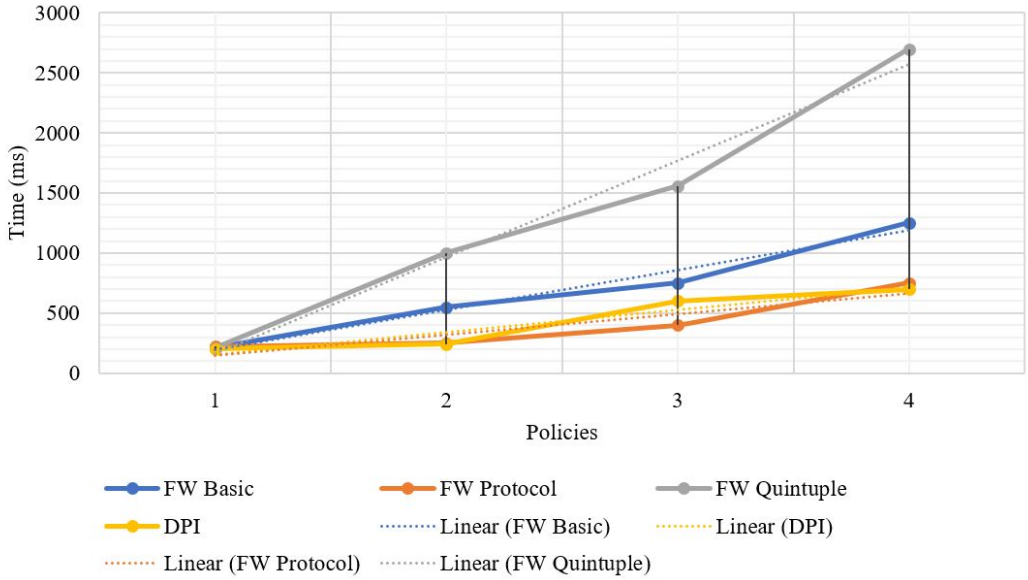


Fig. 6.8 Performance comparison with a single network function

As shown in chart 6.8, firewall and DPI have comparable performance when they both check an integer field. The performance of the firewall begins to degrade when the field is an EnumSort to the point where the firewall has serious scalability issues when a DatatypeSort is used (the wildcards autoconfiguration in the previous chart). In order to limit the scalability problem some modifications have been made on the behavioural model of the involved VNFs. In particular, these modifications improved the VNF formulas that were using the existential quantifier $\exists$ when referring to some neighbour in a *send* or *recv* function. For example, for a firewall:

$$\begin{aligned} &\forall\{next, p0\}: \\ &\qquad send(firewall, next, p0) \implies \\ &\qquad\quad \exists\{previous\} | recv(previous, firewall, p0) \wedge \neg rule \end{aligned} \tag{6.14}$$

To resolve this kind of pattern, z3 uses the skolemization in order to work only with universally quantified formulas. In fact, the skolemization is a special procedure that substitutes every existentially quantified variable with a free function $f$ based on the variable quantified by the universal quantifier that precedes the existential quantifier. In order to elaborate more on this, we show the following formulas :

$$\begin{gathered}\forall x\,[someConditionOn(x) \wedge \exists\{y\}|someOtherConditionOn(y)] \\ becomes \\ \forall x\,[someConditionOn(x) \wedge someOtherConditionOn(f(x))]\end{gathered} \tag{6.15}$$

Resolving this type of formulas can be heavy with bad impact on performance.
Considering formula (6.14), since in a graph the neighbours of a node are known, the existential quantifier can be replaced by the enumeration of the neighbours. Currently in Verifoo there is no distinction between previous nodes and next nodes, therefore in the enumeration all the neighboursa re listed. This does not create any inconsistencies in the results because of how the network behaviour is modeled (i.e. the formulas introduced by the network behaviour will not allow to deliver a packet to a specific end-point, forwarding the packet backwards in a SG, thus avoiding self loops). With this modification, the formula becomes:

$$\begin{gathered}\forall p0 : \\ \left[\bigvee_{\forall i|n_i \in N} send(firewall, n_i, p0)\right] \implies \\ \left[\bigvee_{\forall j|n_j \in N} recv(n_j, firewall, p0)\right] \wedge \neg rule\end{gathered} \tag{6.16}$$

where $N$ is the set of all the neighbours of the firewall. The performance before the modifications can be seen in Figure 6.9, while the changes introduced by them can be observed in the Figure 6.10. The performance has been evaluated with the simpler type of autoconfiguration, the BASIC one which is composed only by IPs and there is no possibility of using the wildcards. As can be seen, the modifications have halved the computational cost for the more complex scenario (more firewall and more policies), but the improvement effect fades in the simpler one. In conclusion, the performance tests executed on the final solution showed that the extended network model for service graphs and the new constraints add a

reasonable amount of computational time. However, the achieved result is satisfying, considering that it would be less than the time needed in a manual configuration and that this automated approach would avoid any human error.

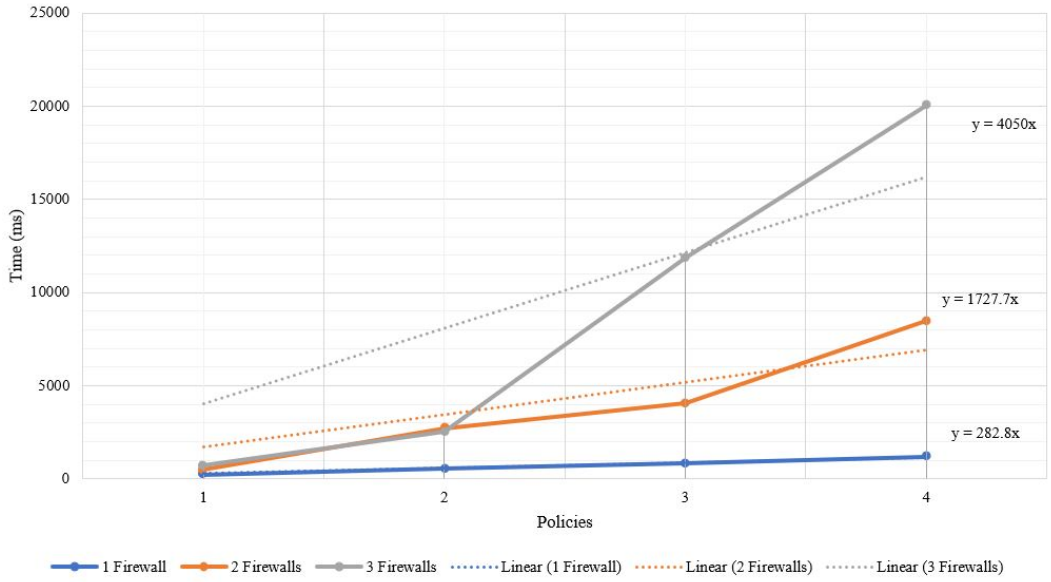


Fig. 6.9 Firewall with BASIC autoconfiguration and no modifications

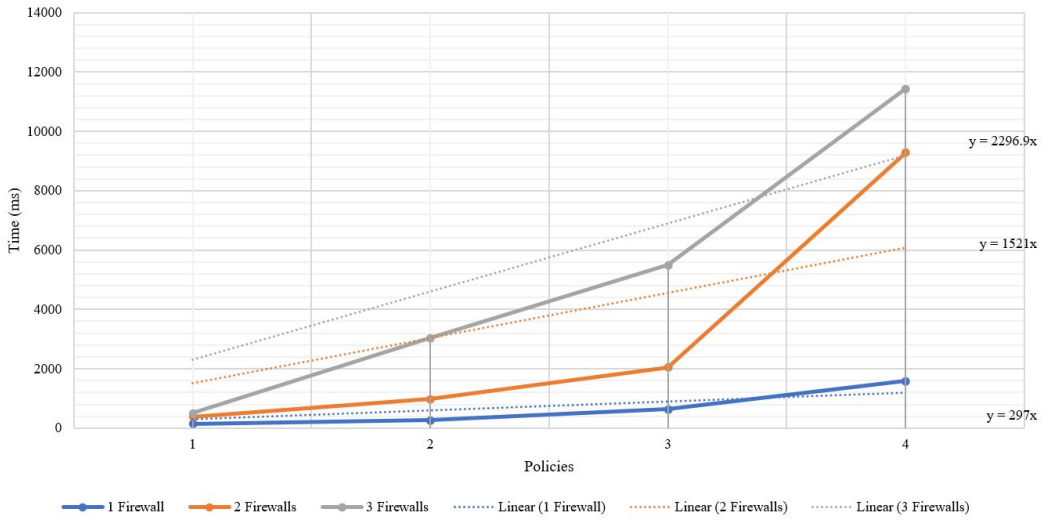


Fig. 6.10 Firewall with BASIC autoconfiguration and modifications

# Chapter 7

# Conclusion

In order to meet the data processing and real-time computing needs of telco operators, distributed computing, distributed storage, microservice technology, and various open source technologies need to be introduced to reduce business support costs.

Nowadays operators propose dynamic network services, where tenants are provided with a particular degree of flexibility to satisfy their requirements. This allows users to construct their own virtual services, chaining, and allocating the required network functions/services in an optimized manner. Providers do not want to pose limitations also on network function selection (i.e., they would like to allow users to choose the optimal number of network functions either from the catalog offered by providers or implemented by third-parties) and are looking for ways to enable further network services (e.g., network automation).

Currently, multiple stakeholders are involved in the development and standardization of these technologies for network softwarization and their embodiment into next-generation networks (e.g., 5G) based on SDN, NFV, and Orchestration building blocks and reference architectures. One of the enablers of this standardization initiative is the ETSI Industry Specification Group regarding the NFV domain, aiming at specifying a reference architecture. However, the scope of these efforts is rather limited, and there is still a need for security checking, to achieve elastic scaling of resources, to improve resource utilization and automatic isolation of faulty devices to ensure system stability.

In this thesis, we focused on different aspects of a network service life cycle. We started by defining a “user-friendly” network function modeling approach, which

allows different vendors to represent an abstract behavior of their network devices. This, in turn, allows them to use various formal analysis tools to verify, prove, and modify network device behaviors and configurations without requiring strong expertise in formal methods.

After generalizing the applicability of the modeling approach to the various network verification tools, we presented a novel approach of joint optimization and verification on the basis of the network function models. In particular, we formulated the optimization and verification problems through the use of MaxSAT, which requires as an input: (i) abstract models of network functions representing both their forwarding behavior and their configuration parameters and the interconnections among functions, (ii) a model of the substrate network, and (iii) the network security properties that must be satisfied. Given these inputs, it generates a formally verified optimal placement plan. Even though the two problems of placement and of formal verification are widely covered separately in the existing literature, to the best of our knowledge, our approach is the only existing one that solves both problems in "one shot" by merging these two concepts together. This approach promises considerable benefits compared to widely known techniques for the virtual network embedding problem. For the first time, we are able to encode expressive constraints such as forwarding behavior of the network, configuration parameters of network functions, and a wide range of security properties in solving an optimization problem. Moreover, the framework developed on the basis of this methodology suggests an optimal placement plan for the network functions in order to respect predefined specifications.

Finally, we have extended this joint analysis approach to automatically define the optimal allocation scheme and configuration of network function instances by refining a service graph provided by the service designer, with respect to security requirements. Our main goal is to provide high confidence that the intended network security policies are correctly and optimally enforced. After having defined a general framework to achieve this goal, we focused attention on a specific class of security functions, i.e. packet filtering firewalls. We developed a procedure for the automatic optimal allocation and configuration of these functions inside a user-provided service graph which provides formal assurance that some user-provided security requirements are satisfied. If requirements are not met, the user receives UNSAT result. Optimality is achieved minimizing the number of allocated firewall instances in the service graph, in order to minimize the amount of resources needed

to deploy the VNFs in the remote servers of a virtualized infrastructure, and by minimizing the number of rules inside each firewall. The latter allows reducing the memory required to store the rules and, whilst, to improve the performance of the VNFs. The presented approach suits the work of a service designer, replacing manual tasks, and contributes to achieving a correct configuration of a network service, by means of its formal approach. At the same time, the approach finds an optimal solution among all the possible ones. Our purpose for a near future is to further refine the methodology, addressing the automatic allocation and configuration of other security functions, such as web application firewalls, anti-spam filters and VPN gateways in a logical service graph. Besides, we are planning to improve the performance, by pursuing a trade-off between optimality of configurations and required computational complexity.

Glossary of terms

| | |
|---|---|
| SDN | Software Defined Networking |
| NFV | Network Function Virtualization |
| VNF | Virtual network functions |
| MaxSMT | Maximum satisfiability modulo theory problem |
| SR | Service requests |
| SG | Service graph |
| SFC | Service function chain |
| NFVO | NFV orchestrator |
| MANO | NFV Management and Orchestration |
| VNFM | VNF manager |
| VIM | Virtual infrastructure manager |
| VNE | Virtual Network Embedding |
| FOL | First-Order Logic |
| SAT | Satisfiability |
| ACL | Access Control List |
| NSD | Network Service Descriptor |
| AST | Abstract Syntax Tree |
| MIQCP | Mixed Integer Quadratically Constrained Programming |
| NSF | Network Security Functions |
| NSR | Network Security Requirements |
| AG | Allocation Graph |
| BBU | Baseband Unit |
| C-RAN | Centralized, Collaborative, Cloud and Clean RAN |
| CU | Central Unit |
| DU | Distributed Unit |
| eMBB | Enhanced Mobile Broadband |
| PDCP | Packet Data Convergence Protocol |
| RLC | Radio Link Control |
| RRC | Radio Resource Control |
| uRLLC | Ultra-reliable low latency communication |

# References


[1] M. M. Hasan and H. T. Mouftah. Cloud-centric collaborative security service placement for advanced metering infrastructures. *IEEE Tran. on Smart Grid*, PP(99), 2017.

[2] Network function virtualization - White Paper 2. Technical report, The European Telecommunications Standards Institute, October 2013.

[3] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN. *Queue*, 11(12):20–40, dec 2013.

[4] Network Functions Virtualisation (NFV): Management and Orchestration (GS/NFV-MAN-001). Technical report, The European Telecommunications Standards Institute, 2014.

[5] Vision on Software Networks and 5G. 5g ppp architecture working group.

[6] J. Waryet [13] P. Bisson. 5g ppp phase1 security landscape.

[7] Network Functions Virtualisation (NFV) Ecosystem. Report on sdn usage in nfv architectural framework.

[8] OASIS. Tosca simple profile in yaml. White paper, 2014. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf.

[9] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996.

[10] Serena Spinoso, Matteo Virgilio, Wolfgang John, Antonio Manzalini, Guido Marchetto, and Riccardo Sisto. Formal Verification of Virtual Network Function Graphs in an SP-DevOps Context. In *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, pages 253–262, 2015.

[11] F. Valenza, S. Spinoso, C. Basile, R. Sisto, and A. Lioy. A formal model of network policy analysis. In *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 516–522, Sep. 2015.

[12] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, 2016. USENIX Association.

[13] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA, 2015. USENIX Association.

[14] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[15] Brendan Tschaen, Theophilus Benson Ying Zhang, Jeongkeun Lee Sujata Banerjee, and Joon-Myung Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016.

[16] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying isolation properties in the presence of middleboxes. *CoRR*, abs/1409.7687, 2014.

[17] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. *ACM SIGPLAN Notices*, 49(6):282–293, jun 2014.

[18] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Scalable Symbolic Execution for Modern Networks. *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 314–327, 2016.

[19] Sanjai Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Large Installation System Administration Conference - Volume 19*, LISA '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[20] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 101–114, Seattle, WA, April 2014. USENIX Association.

[21] Shuhao Liu, Zhiping Cai, Hong Xu, and Ming Xu. Towards security-aware virtual network embedding. *Computer Networks*, 91:151 – 163, 2015.

[22] Andreas Fischer and Hermann Meer. Position paper: Secure virtual network embedding. *PIK - Praxis der Informationsverarbeitung und Kommunikation*, 34, 01 2011.

[23] Etsi Gs Nfv 001 V1.1.1. Network Functions Virtualisation (NFV); Terminology. *IEEE Network*, 1(5):1–50, 2013.

[24] "Study on new radio access technologies: Radio access architecture and interfaces. Technical report, 3GPP TR 38.801 v14.0.0, 2017.

[25] Amar Kapadia and Nicholas Chase. *Understanding OPNFV: Accelerate NFV Transformation Using OPNFV*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 1st edition, 2017.

[26] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: Toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55:38–42, 10 2012.

[27] Giuseppe Antonio Carella and Thomas Magedanz. Open baton: A framework for virtual network function management and orchestration for emerging software-based 5g networks. *Newsletter*, 2016, 2015.

[28] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys Tutorials*, 15(4):1888–1906, Fourth 2013.

[29] Y. Zhu and M. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12, April 2006.

[30] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[31] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. *SIGPLAN Not.*, 52(1):572–585, January 2017.

[32] Robert Soulé, Shrutarshi Basu, Parisa Jalili, Fernando Pedone, Robert Kleinberg, Emin Sirer, and Nate Foster. Merlin: A language for provisioning network resources. *CoNEXT 2014 - Proceedings of the 2014 Conference on Emerging Networking Experiments and Technologies*, 07 2014.

[33] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of ip networks. In *INFOCOM*, pages 2170–2183. IEEE, 2005.

[34] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Cataldo Basile, Daniele Canavese, Christian Pitscheider, Antonio Lioy, and Fulvio Valenza. Assessing network authorization policies via reachability analysis. *Comput. Electr. Eng.*, 64(C):110–131, 2017.

[36] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 29–43, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Y. Yang, K. McLaughlin, S. Sezer, Y. B. Yuan, and W. Huang. Stateful intrusion detection for iec 60870-5-104 scada security. In *2014 IEEE PES General Meeting | Conference Exposition*, pages 1–5, 2014.

[38] Joaquin Garcia-Alfaro, Frédéric Cuppens, Nora Cuppens-Boulahia, Salvador Martinez, and Jordi Cabot. Management of stateful firewall misconfiguration. *Computers Security*, 39:64 – 85, 2013. 27th IFIP International Information Security Conference.

[39] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[40] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[41] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 29–35, New York, NY, USA, 2016. ACM.

[42] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 239–253, Berkeley, CA, USA, 2016. USENIX Association.

[43] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[44] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[45] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[46] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth Krishnamurthy, Kevin Chan, and Tracy Braun. Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery. 01 2020.

[47] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Computer Architecture News*, 39, 06 2012.

[48] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. Wilfong, Y. R. Yang, and C. Guo. Pace: Policy-aware application cloud embedding. In *2013 Proceedings IEEE INFOCOM*, pages 638–646, April 2013.

[49] X. Li and C. Qian. An nfv orchestration framework for interference-free policy enforcement. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 649–658, June 2016.

[50] A. H. M. Jakaria, M. A. Rahman, and C. Fung. A requirement-oriented design of nfv topology by formal synthesis. *IEEE Transactions on Network and Service Management*, 16(4):1739–1753, Dec 2019.

[51] E Hadjiconstantinou. Transformation of propositional calculus statements into integer and mixed integer programs: An approach towards automatic reformulation. *Brunel University Mathematics Technical Papers collection;*, 1990.

[52] Shuhao Liu, Zhiping Cai, Hong Xu, and Ming Xu. Towards security-aware virtual network embedding. *Computer Networks*, 91(Supplement C):151 – 163, 2015.

[53] 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)*, pages 378–384, Oct 2012.

[54] Byoung-Moon Chin, Young-Han Choe, Sung-Un Kim, and Jae-II Jung. Automated test generation from specifications based on formal description techniques. *ETRI Journal*, 19(4):363–388, 1997.

[55] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, November 2004.

[56] Pavan Verma and Atul Prakash. FACE: A firewall analysis and configuration engine. In *2005 IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2005), 31 January - 4 February 2005, Trento, Italy*, pages 74–81, 2005.

[57] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *1997 IEEE Symposium on Security and Privacy, May 4-7, 1997, Oakland, CA, USA*, pages 120–129, 1997.

[58] Nihel Ben Youssef and Adel Bouhoula. A fully automatic approach for fixing firewall misconfigurations. In *11th IEEE International Conference on Computer and Information Technology, CIT 2011, Pafos, Cyprus, 31 August-2 September 2011*, pages 461–466, 2011.

[59] Kamel Adi, Lamia Hamza, and Liviu Pene. Automatic security policy enforcement in computer systems. *Computers & Security*, 73:156–171, 2018.

[60] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 359–373, 2017.

[61] Cataldo Basile, Fulvio Valenza, Antonio Lioy, Diego R. Lopez, and Antonio Pastor Perales. Adding support for automatic enforcement of security policies in NFV networks. *IEEE/ACM Trans. Netw.*, 27(2):707–720, 2019.

[62] Eder J. Scheid, Cristian Cleder Machado, Muriel Figueredo Franco, Ricardo Luis dos Santos, Ricardo J. Pfitscher, Alberto E. Schaeffer Filho, and Lisandro Zambenedetti Granville. Inspire: Integrated nfv-based intent refinement environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon, Portugal, May 8-12, 2017*, pages 186–194, 2017.

[63] Yoonseon Han, Jian Li, Doan Hoang, Jae-Hyoung Yoo, and James Won-Ki Hong. An intent-based network virtualization platform for SDN. In *12th International Conference on Network and Service Management, CNSM 2016, Montreal, QC, Canada, October 31 - Nov. 4, 2016*, pages 353–358, 2016.

[64] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining network intents for self-driving networks. *Computer Communication Review*, 48(5):55–63, 2018.

[65] Zheng Hao, Zhaowen Lin, and Ran Li. A sdn/nfv security protection architecture with a function composition algorithm based on trie. In *Proc. of the 2Nd International Conference on Computer Science and Application Engineering*, CSAE '18, pages 176:1–176:8, 2018.

[66] MyungKeun Yoon, Shigang Chen, and Zhan Zhang. Minimizing the maximum firewall rule set in a network with multiple firewalls. *IEEE Trans. Computers*, 59(2):218–230, 2010.

[67] Mohammad Ashiqur Rahman and Ehab Al-Shaer. Automated synthesis of distributed network access controls: A formal framework with refinement. *IEEE Trans. Parallel Distrib. Syst.*, 28(2):416–430, 2017.

[68] Wenyu Shen, Masahiro Yoshida, Kenji Minato, and Wataru Imajuku. vconductor: An enabler for achieving virtual network integration as a service. *IEEE Communications Magazine*, 53(2):116–124, 2015.

[69] Serena Spinoso, Marco Leogrande, Fulvio Risso, Sushil Singh, and Riccardo Sisto. Seamless configuration of virtual network functions in data center provider networks. *J. Network Syst. Manage.*, 26(1):222–249, 2018.

[70] Kostas Giotis, Yiannos Kryftis, and Vasilis Maglaris. Policy-based orchestration of NFV services in software-defined networks. In *Proc. of the 1st IEEE Conference on Network Softwarization, NetSoft 2015, London, United Kingdom, April 13-17, 2015*, pages 1–5, 2015.

[71] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK, 1992. Springer-Verlag.

[72] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[73] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[74] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1–3):153–183, June 2005.

[75] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 269–282, New York, NY, USA, 1979. Association for Computing Machinery.

[76] Serena Spinoso, Matteo Virgilio, Wolfgang John, Antonio Manzalini, Guido Marchetto, and Riccardo Sisto. Formal verification of virtual network function graphs in an sp-devops context. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service Oriented and Cloud Computing*, pages 253–262, Cham, 2015. Springer International Publishing.

[77] Herbert B Enderton. *A mathematical introduction to logic*, volume 40, page 317. 2001.

[78] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA, 2017. USENIX Association.

[79] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *SIGCOMM Comput. Commun. Rev.*, 38(2):17–29, March 2008.

[80] Xiang Cheng, Sen Su, Zhongbao Zhang, Kai Shuang, Fangchun Yang, Yan Luo, and Jie Wang. Virtual network embedding through topology awareness and optimization. *Computer Networks*, 56(6):1797 – 1813, 2012.

[81] Jalolliddin Yusupov Guido Marchetto, Riccardo Sisto and Adlen Ksentini. Formally verified latency-aware vnf placement in industrial internet of things. In *14th IEEE International Workshop on Factory Communication Systems (WFCS), Imperia, Italy*, 2018. In press.

[82] Joao Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3):317–343, 2011.

[83] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. SNDlib 1.0–Survivable Network Design Library. In *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, April 2007. http://sndlib.zib.de, extended version accepted in Networks, 2009.

[84] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[85] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

[86] Tiago Camilo, Jorge Sá Silva, André Rodrigues, and Fernando Boavida. Gensen: A topology generator for real wireless sensor networks deployment. In Roman Obermaisser, Yunmook Nah, Peter Puschner, and Franz J. Rammig, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 436–445, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[87] Arsany Basta, Wolfgang Kellerer, Marco Hoffmann, Hans Jochen Morper, and Klaus Hoffmann. Applying nfv and sdn to lte mobile core gateways, the functions placement problem. In *Proc. of the 4th ACM Workshop on All Things Cellular: Operations, Applications, & Challenges*, 2014.

[88] NGMN Alliance. 5g end-to-end architecture framework. Technical report, 2017. 04-Oct.

[89] Verizon. Data Breach Investigations Report, 2019.

[90] Kresimir Popovic and Zeljko Hocenski. Cloud computing security issues and challenges. pages 344 – 349, 06 2010.

[91] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1):236–262, 2016.

[92] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, 2015.

[93] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Steven Latré, Marinos Charalambides, and Diego López. Management and orchestration challenges in network functions virtualization. *IEEE Communications Magazine*, 54(1):98–105, 2016.

[94] Jiao Zhang, Zenan Wang, Ningning Ma, Tao Huang, and Yunjie Liu. Enabling efficient service function chaining by integrating NFV and SDN: architecture, challenges and opportunities. *IEEE Network*, 32(6):152–159, 2018.

[95] Ehab Al-Shaer, Hazem H. Hamed, Raouf Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications*, 23(10):2069–2084, 2005.

[96] F. Valenza, C. Basile, D. Canavese, and A. Lioy. Classification and analysis of communication protection policy anomalies. *IEEE/ACM Transactions on Networking*, 25(5):2601–2614, Oct 2017.

[97] Cataldo Basile, Daniele Canavese, Antonio Lioy, and Fulvio Valenza. Inter-technology conflict analysis for communication protection policies. In Javier Lopez, Indrajit Ray, and Bruno Crispo, editors, *Risks and Security of Internet and Systems*, pages 148–163, Cham, 2015. Springer International Publishing.

[98] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[99] EU research innovation programme. Addressing threats for virtualised services, 2018.