POLITECNICO
DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Automatic Optimized Firewalls Orchestration and Configuration in NFV environment

Supervisors
prof. Riccardo Sisto
prof. Guido Marchetto
dott. Fulvio Valenza
dott. Jalolliddin Yusupov

Candidate
Daniele Bringhenti

Academic Year 2018-2019

This work is subject to the Creative Commons Licence

Summary

The *Network Functions Virtualization* (NFV) paradigm is a novel networking technology which, by means of a decoupling between the network functions and the hardware appliances, allows software processes to be installed as service functions on general-purpose servers. Among the consequent benefits, this principle entails further agility and flexibility in the creation of a *Service Graph*, which is a generalization of the *Service Function Chain* (SFC) concept, describing how the single network functions, needed to create a complete end-to-end service, must be organized and connected.

A problem which, however, arises in the creation of a Service Graph in this scenario is that this task is typically performed by a service designer; instead, the security manager is separately in charge of the allocation and configuration of the *Network Security Functions* (NSFs) – such as firewalls and anti-spam filters – needed to protect the network from cybersecurity attacks. Moreover, these operations are usually performed manually, so they are prone to human errors and the reaction latency is not negligible whenever the security defences should be updated according to different or additional security requirements.

In view of these considerations, this thesis contributed to the development of *VEREFOO* (VErified REFinement and Optimized Orchestration), a framework which aims to provide a *Security Automation* approach as a solution to these open problems. The main purpose is to perform, on a provided Service Graph, an automatic optimized allocation and configuration of the NSFs that are necessary to fulfil an input set of Network Security Requirements, which can be expressed by the service designer by exploiting a high-level language. The VEREFOO approach involves the formulation of a MaxSMT problem, whose objectives are to satisfy a set of hard constraints that always require to be fulfilled and, at the same time, to achieve the maximum sum of the specific weights that the soft clauses are given. Its targets are on one side the allocation of the minimum number of NSFs instances to reduce the resource consumption due to the allocation of the corresponding virtual functions, on the other side the reduction of the rules describing their configuration to improve the efficiency of the filtering operations. The MaxSMT problem is formulated so as to provide also a formal verification that the achieved solution is formally correct.

The major contributions provided by this thesis work have been the formal definition of the optimization and verification problem and its implementation by means of z3, a state-of-the-art MaxSMT solver, inside the framework. Among all the possible NSFs, the focus has been on packet filter, the most common firewall technology which can filter the received packets according to the values of the IP

quintuple. An automatic generation of both the allocation schema and the filtering policies of the firewalls is, currently, an open problem not well addressed in literature by itself. Hence, the solution developed in this thesis advances the state of the art. In order to make this solution really effective, it has been necessary to develop a number of pruning strategies to minimize the number and the complexity of the clauses that define the MaxSMT problem.

The implementation has been finally tested extensively in common network scenarios and it showed good scalability against the dimension of the Service Graph and the number of input security requirements; consequently, this thesis demonstrates that the proposed approach is feasible and that it can provide a valid alternative in enforcing security functions to manual allocation and configuration of packet filtering firewalls, enabling low latency reaction to changes in security requirements. Furthermore, the approach has been developed to be compliant with future extensions such as the support of other NSFs in order to enrich its capabilities.

Acknowledgements

This thesis represents the conclusions of a five-year path, where I was able not only to enrich my technical background, but also to understand what are my main interests and which objectives I should pursue accordingly to them.

First of all, I want to acknowledge the supervisors of my thesis. I thank the professors who supervised my thesis work, that are professors Sisto and Marchetto: they gave me the possibility to work on a new research path, that is full of potentialities and could really contribute to improve the state-of-the-art of the networking field. They were always available to listen to what I had achieved and to provide me their valuable feedback, helping me to improve in different aspects. I also want to thank Fulvio for all the suggestions he provided me, for his availability even when he was not in Turin, for sharing his research experience. Then, I have to thank Jalol for being always available every day, for all the time he spent to help me to achieve the results of this thesis, for all his daily tips which greatly helped the work.

Then, I want to acknowledge all the people – my family, my friends and all the others – who were alongside me in this path. Among all of them, I cannot not mention my parents and, in particular, my mother: she always helped me, always listened to me even though she sometimes (i.e. often) did not understand what I was telling her when asking for her suggestion and, I am sure, will be always on my side. Then, a thought is for my grandfather, who would have really wanted to be with me these days.

A path is finishing, but it could be just the beginning of a new one. And, about this, I want again to thank professor Sisto, for the possibility he is giving me, for assisting me in the writing of my first research papers and for all the support in this new path I could potentially follow.

At the moment I am writing these acknowledgements I still do not if I will be granted to follow this new path, but what I am sure is that *no one knows what the future holds, that is why its potential is infinite.*

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Thesis objective

In the recent years, novel networking technologies have emerged; in particular, the *Network Functions Virtualization* (NFV) principle allows an agile deployment of network functions required to provide a service and the *Software-Defined Networks* (SDN) paradigm introduces the possibility to create a forwarding path by means of a software process. In a scenario characterized by these emerging technologies, new open research paths can be followed in the next-generation computer networks and in the cybersecurity context.

If in the traditional approach exploited to create a Service Graph – that is a combination of single service functions – there was the necessity to install specific hardware appliances to provide service features like web caching and load balancing, thanks to these novel networking technologies in the future it could become common to virtualize the network functions. Virtual functions, actually, can be deployed dynamically on servers and switched on every time they are needed, while they can be turned off or their configuration can be changed when different requirements are provided by the service designer. Virtualization and automation are becoming key terms in both data centers and *Service Provider* (SP) networks, where agility and capability of reacting to events are fundamental factors in order to be compliant with the scalability of these big scenarios.

Moreover, these concepts can be applied not only to network service functions like the web cache and the *Network Address Translator* (NAT), but also to *Network Security Functions* (NSFs) like the packet filter firewall and the *Intrusion Detection System* (IDS); actually, they can ideally be deployed as virtual services on-demand, providing a faster reaction to incoming attacks. In addition, to unburden the workload of the service designer, their configuration could be automatized by means of a distribution process, which has the task to create appropriate policy rules to satisfy the *Network Security Requirements* (NSRs) representing the security constraints to be compliant with, such as the need of isolation between a server and a group of hosts.

The problem which arises, however, is that currently an automatic process to select the security functions to deploy in order to satisfy a set of high-level Network

Security Requirements does not exist [1]; instead, the service designer must decide which are the best functions to use without often using optimal criteria but simple heuristics or *try-and-error* approach. The same consideration can be applied to the configuration of the selected functions; if auto-configuration of network functions has been researched in traditional network scenarios, like the *Firmato* [2] toolkit did for traditional firewalls, in the NFV context the research is now making the first steps in establishing a method to automatically configure the policy rules.

VEREFOO (VErified REFinement and Optimized Orchestration) is a novel framework which aims to perform a policy refinement (i.e. translation of high-level requirements into medium-level expressions), an optimal allocation of the Network Security Functions required to satisfy these security constraints in a logical topology, an optimal distribution of the policy rules on the allocated functions and, finally, an optimal deployment of the virtualized functions on a substrate network. For these purposes, it exploits *z3*, a theorem solver developed by Microsoft Research, and *Verigraph*, a framework developed in Politecnico di Torino for the requirements verification in a *Virtual Network Embedding* (VNE) scenario.
The benefit which in the future this approach could provide is an automatic way to configure the network and periodically check if the configuration is compliant with the security requirements, every time a new constraint is specified or an existing one is modified.

The thesis objective has been to contribute to the design, implementation and refinement of an existing framework, with the goal to create the *ADP* (Allocation, Distribution, Placement) module of VEREFOO, in charge of solving a MaxSMT optimization problem related to the automatic allocation of Network Security Functions and configuration of the policy rules. The most relevant goals have been to extend the usage scenarios and to improve the performance of this module; consequently, new different ways to use the z3 language have been pursued, new features like the Allocation Graph have been implemented and a further abstraction level between the creation of routing tables and the specific behaviour of the network functions has been introduced. Among the Network Security Functions which the ADP module can allocate and configure, the focus has been on the packet filter firewall, to provide a complete and functional example of how the security requirements specified by the service designer can be distributed in multiple instances of this firewall technology, while in the meantime establishing their optimal allocation on the network topology.

1.2 Thesis description

After **Chapter 1** briefly introduced the problems to challenge and the goals to achieve, the rest of the thesis is structured in the following way:

- **Chapter 2** describes the novel networking paradigms representing the scenario of the thesis work, i.e. the *Software-Defined Networks* (SDN) and the *Network Functions Virtualization* (NFV) principles; the focus is on the relevant details of the possible architectural solutions and the benefits which they

can bring to the definition of a service through a Service Graph, generalization of the *Service Function Chain* concept.

- **Chapter 3** describes the concept of policy-based management in distributed systems as a method to formally present a set of goals which the *Network Security Functions* must reach by means of their behaviour and configuration. After briefly delineating the basic architectural model of a policy-based framework, the chapter focuses on one hand on the policy specification and abstraction, providing an overview of the policy hierarchy and the possible modelling languages, on the other hand on the processes of policy refinement and translation, whose purpose is to move between the policy abstraction levels. Finally, some relevant related works about firewall auto-configuration are presented, such as *Firmato*, a firewall management toolkit which inspired the auto-configuration model of packet filters designed in the thesis work.

- **Chapter 4** describes the main tools used for the development of the framework: z3, which is a theorem prover from Microsoft Research used to solve a MaxSMT problem, and Verigraph, whose purpose is the verification of security requirements in a network characterized by functions deployed in a virtualized environment.

- **Chapter 5** provides a complete description of the workflow of VEREFOO (VErified REFinement and Optimized Orchestration), with a major focus on the ADP (*Allocation, Distribution, Placement*) element on which the development presented in this thesis mostly concentrated. Usage scenarios are presented to describe how the framework can be used by a service designer for the allocation of *Network Security Functions* on a logical topology - the Allocation Graph - derived from the definition of a Service Graph, the distribution of the policy rules in the allocated functions and their placement in the physical infrastructure.

- **Chapter 6** describes the proposed model and implementation of the Allocation Graph (i.e. the logical topology on which the *Network Security Functions* can be allocated by the ADP element of VEREFOO), the first-order logic formulas and the corresponding implementation in a z3 model of the forwarding rules by means of which the packets can transit in the network.

- **Chapter 7** describes the first-order logic formulas and the implementation of the medium-level reachability and isolation security requirements, which respectively require that a packet sent by a source end point must reach the destination or should be blocked by an intermediate middlebox in the network. The main extensions are highlighted, alongside with the modifications of the *wildcards* feature, which increases the expressiveness of the IP addresses by adding a partial model of the netmask concept inside the data structures representing the addresses themselves.

- **Chapter 8** describes the packet filter, the firewall technology on which the thesis focuses with the goal to present a proof of concept for the allocation and distribution operations in the logical topology. In this chapter a complete overview of its formal model and implementation is provided, with an accurate

description of the soft and hard constraints introduced in the optimization problem.

- **Chapter 9** presents the results of some performance tests which were carried out to show how the new framework version is able to get better performance than the original one, to understand which goals have been achieved, which are the current limitations and which further aspects should be addressed in the future.
- **Chapter 10** summarizes which goals this thesis work succeeded in reaching, what are the main research directions that could be followed to improve the implementation and which features could be introduced to enrich the capabilities of the framework.
- **Appendix A** presents a manual which future developers and maintainers of the VEREFOO code can exploit to understand how to use the state-of-the-art z3 MaxSMT solver in the Java language.
- **Appendix B** describes the design of REST-based APIs to interface with the ADP module of the framework and the implementation of the corresponding RESTful web service.

Chapter 2

Software Networking

This chapter introduces the main concept of *Service Function Chain* (SFC), which provides an abstract description of a complete end-to-end service characterized by single network functions and by their ordering constraints; this concept represents the basic theory for the creation of a Service Graph, which can be considered a generalization of a linear SFC. Furthermore, the main limitations of the traditional deployment approach of an SFC on hardware appliances is briefly illustrated to clarify which problems must be challenged in this scenario.

Then, with the purpose to show how the limits can be overcome, novel technologies which have recently introduced new research challenges in the networking field are presented; the *Software-Defined Networks* (SDN) and the *Networks Functions Virtualization* (NFV) put a strong emphasis on the main features of software, which are agility in the deployment, the ease of re-programmability every time a modification in the network is needed and the customization of the services for the different users.

Finally, a brief overview about the principles of *Network Automation* is highlighted to show the benefits of an approach where the configuration of a Service Function Chain is based not only on the user's commands, but also on reactions triggered by events coming from the network itself.

2.1 Service Function Chain

The creation of a complete end-to-end network service usually requires a set of functions, which should be applied on a specific sequence to the traffic in order to achieve a result which satisfies the service designer's needs or the user's requests. The concept of *Service Function Chain* (SFC), corresponding to the description of the complete service, has been formally presented in the RFC 7665 [3] from which the following definitions come:

Service Function (SF) It is a function that is responsible for specific treatment of received packets. A Service Function can act at various layers of a protocol stack (e.g. at the network layer or other OSI layers). As a logical component, a service function can be realized as a virtual element or be embedded in a

physical network element. One or more Service Functions can be embedded in the same network element. Multiple occurrences of the service function can exist in the same administrative domain.

Service Function Chain (SFC) A Service Function Chain defines an ordered set of abstract service functions and ordering constraints that must be applied to packets and/or frames and/or flows selected as a result of classification. The implied order may not be a linear progression as the architecture allows for SFCs that copy to more than one branch, and also allows for cases where there is flexibility in the order in which service functions need to be applied. The term "service chain" is often used as shorthand for Service Function Chain.

In Figure 2.1, for example, an end-to-end service between a web client and a web server is characterized by a specific sequence of elementary blocks: the first one is a firewall which drops most of the unwanted traffic, the second one is an *Intrusion Detection System* (IDS) which can detect attacks based on patterns or a previous monitoring phase and finally the third one is a reverse proxy whose main purposes are obfuscation of the real characteristics of the server, web caching of static contents and spoon-feeding. It is evident how these functions should be applied to the packets in the path from the client to the server in this specified order to compensate the relative weaknesses, e.g. the IDS can detect attacks that passed through the firewall, overcoming its potential implementation flaws or limitations.

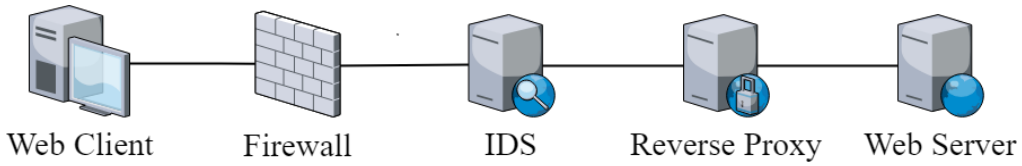


Figure 2.1. Example of a Service Function Chain

In the past, in a traditional scenario like the edge of a Service Provider network or the data center of a Content Provider company, network functions which should provide a service to a customer were mainly based on hardware appliances. Each physical middlebox was specifically designed, built and dedicated to implement a corresponding function and a combination of them represented a Service Function Chain; a lower level of abstraction was this way provided.

This original approach is evidently characterized by several limitations:

- sharing resources between network functions is not feasible because each one is a dedicated hardware box, even though this feature would be extremely useful because it would let an unloaded function leave computational resources to another one in sufferance;
- it is not easy to force the traffic to pass though specific elements of the Service Function Chain because in the IP world the traffic is simply forwarded by means of a technique of *routing by network address*;

- personalizing the Service Function Chain for each different user is not trivial because the links between the appliances are fixed;
- modifying the structure of a Service Function Chain with a new function is expensive because it requires the purchase of the dedicated hardware appliance and its installation could take a significant time;
- maintenance of the appliances requires a constant monitoring and work by the provider of the Service Function Chain.

For the aforementioned reasons, the main tendency in the last years has been to move both the construction of a path that a packet must follow and the network functions implementation from the hardware level to the software, with the goal to get better deployment agility and performance.

2.2 Software-Defined Networks

2.2.1 Principles of Software-Defined Networks

A novel approach which can be followed to unburden the limitations, described in Section 2.1, due to a traditional hardware implementation of a Service Function Chain is the *Software-Defined Networks* (SDN) principle [4]. As the name suggests, this technology allows the creation of the paths which the packets must follow inside the physical network by means of a software process.

The pillars of the original SDN theory are three:

1. decoupling between the data plane, which takes decisions about the forwarding and manages the low-level modifications of the packets, and the control plane, which coordinates the forwarding actions building shared data structures like the filtering database for a bridge or a routing table for a router;
2. centralization of all the control plane functions in a unique module, generally called *SDN Controller*, which is the place where all the intelligence of this technology is situated. Technically the centralization could be logical through a physical distribution on collaborating nodes, but most of the actual implementations are based on a physical centralization;
3. definition of southbound and northbound interfaces for the communication between the SDN Controller and, respectively, the forwarding infrastructure and the user-level applications or controllers of higher level.

2.2.2 Architecture of an SDN-based model

Figure 2.2 illustrates the typical architecture of a Software-Defined Networks model.

The network infrastructure is characterized by data forwarding elements which do not necessarily require to be complex routers or vendor-customized devices, but

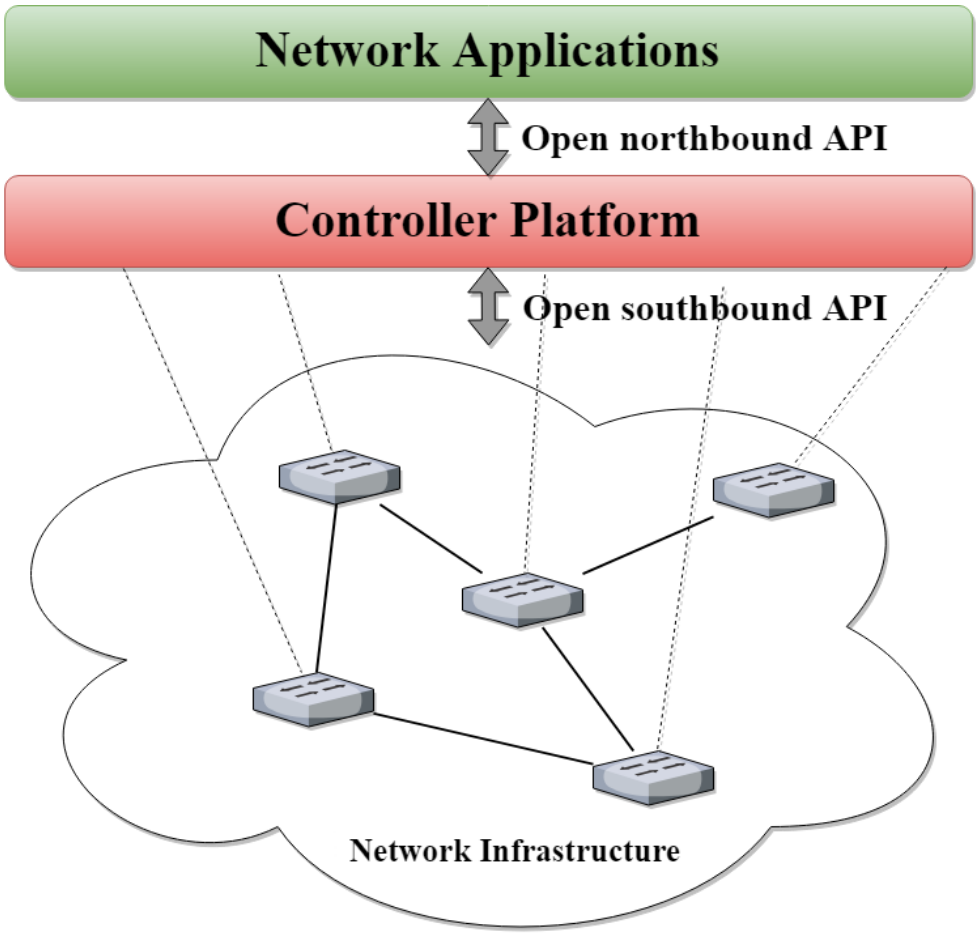


Figure 2.2. Representation of the architecture of a Software-Defined Networks infrastructure

they can be white-label switches; they are built with simple commodities because their only role is to forward the input packets to the correct output ports in the path to reach their final destinations as fast as possible, since efficient forwarding is the key word in the SDN technology. Every switch is then characterized by a forwarding table, with a list of rules; they are made by the *match* fields, which allow to filter only the packets that satisfy them, and the *action* fields, which specify the operations the switch must perform on those packets (e.g. forward to a specific port, forward to the controller platform, forward at a specific rate, drop the packet, push or pop a field, overwrite or modify a header field).

The use of simple forwarding elements is acceptable because their application logic is completely governed by the SDN Controller; through an open southbound interface, it can receive information from the network, like the packets which did not match any rules of the switches, and it creates the required forwarding rules basing the decision on the user's needs and on events coming from the network. Then the controller installs the rules on the switches using southbound protocols like OpenFlow, some REST APIs, *Simple Network Management Protocol* (SNMP), *Network Configuration Protocol* (NETCONF) or RESTCONF.
The fact that a multitude of protocols are available and actually used is motivated

by the legacy hardware already installed in the network; even though OpenFlow was originally proposed as the unique southbound protocol, it would have required a substitution or a software patch for most of the existing elements which did not support it. For this reason, in an SDN Controller a *Service Abstraction Layer* (SAL) is fundamental to abstract the specificity of the drivers in regard to the applications which use them to interact with the switches.

Finally, in addition to traditional applications like the *Topology Manager* and the *Host Tracker* which are directly offered by the Network Operating System of the controller platform, also third-party or external network applications can interact with the overall infrastructure by means of an open northbound interface, which is typically REST-based; the graphical dashboard in this scenario has limited capabilities, such as only showing the results of a monitoring activity, because a greater emphasis is put on the automatic reactions of the controller based on inputs from the software itself. Besides the northbound APIs represent the real added value to the model because they allow the deployment of any high-level network application.

This infrastructure presents the main benefits of a single-point-of-control through which the entire network can be managed and the flexibility in the creation of the forwarding rules, which can be modified according to the actual network behaviour. However, the controller platform can be both a single-point-of-failure in terms of security because it can be accessed by all the physical switches in the forwarding plane and a bottleneck in terms of performance, since the latency with which it replies to the contacting switches is not negligible.

2.2.3 Application of the SDN technology to a SFC

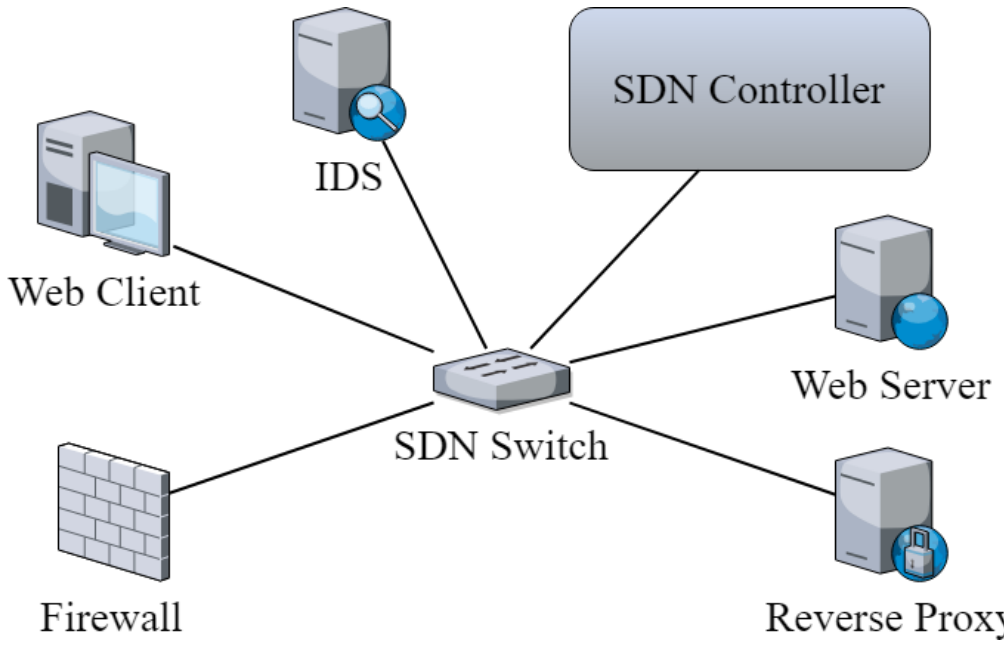


Figure 2.3. Example of Service Function Chain modelled with an SDN architecture.

Supposing that each service function of a Service Function Chain corresponds to a hardware appliance, they can be linked together with a network of SDN switches, whose forwarding behaviour can be completely governed by the controller platform. Figure 2.3 shows how the Service Function Chain represented in Figure 2.2 can be modelled in an SDN architecture; in this example, firewall, IDS and reverse proxy are linked to the switch with a single connection, because the flow of the packets and the crossing order of the middleboxes are determined by the controller platform.

The main advantages this solution provides are, in fact, the following ones:

- it is easy to force the IP traffic to pass through specific appliances thanks to the forwarding rules installed by the controller on the SDN switches, in a proactive way or as reaction to events coming from the network itself;
- changing the configuration of an existing Service Function Chain is relatively fast, e.g. when a new service is added only a link to a switch should be installed and then the rules are generated by a software process;
- it is possible to personalize the different Service Function Chains according to users' needs so that they can share only some of the network functions.

The limitation which cannot be overcome with an infrastructure exclusively based on the SDN technology is, however, that the appliances are still hardware-based, so neither sharing computational resources is feasible nor purchasing and installing a new function are fast operations.

2.3 Network Functions Virtualization

2.3.1 Principles of Network Functions Virtualization

The immediate next step is represented by the concept of *Network Functions Virtualization* (NFV) [5]. The core idea of this novel technology is that each function is a software process which does not require any more dedicated hardware, but it can run on a general-purpose server; so, if SDN focuses on the creation of the forwarding paths by software, NFV targets the virtualization of the *computing*.

A first way to implement virtualized network functions is represented by the traditional Virtual Machines. They are able to provide a strong container isolation and possibility of enforcing operations by the hypervisor, because every time a new Virtual Machine is created, a hardware profile is defined and a different Operating System, with other applications, can be installed on it independently of the hypervisor; it is consequently the best solution for scenarios like a multi-tenant data center, where protection from accesses to their own services is a fundamental issue to challenge.
This benefit is obtained at the expense of high memory and disk space requirements, because of the intrinsic independence from the host; moreover, the live migration of a Virtual Machine is a burdensome task and it requires algorithms which consider

the possibility that a block of memory is modified after being transferred to the new target zone.

Subsequently lightweight virtualization became the new goal to achieve. A first attempt is represented by the *Linux Containers* (LXC), based on the idea of namespaces in the Linux world: the kernel of the hypervisor can be partitioned in respect to one (or more) out of seven possible parameters, so that the created namespace shares all the components of the host kernel related to the specified parameters. For instance, if the network namespace is created, it shares the networking stack with the hypervisor, while all the other parts of the kernel (e.g. the file system) are duplicated and separated.
If the benefit is that the memory occupancy is much lower than what a traditional Virtual Machine requires, the problems are the difficult management of the different parameters of a Linux Container where all the seven kernel components are shared and the absence of the portability feature: in fact, when a LXC is created on a machine, it cannot be migrated to another.

Dockers were the next lightweight virtualization entity which was proposed and, despite the principles on which they are based are the same as for Linux Containers, their success was established by their portability; after a Docker is created, it can be transferred to another machine by means of an intermediate repository (e.g. Docker Hub) on which it is pushed or it can be reproduced using a Docker File, containing a list of operations that must be performed to achieve the same Docker. Also the potential problem about the separated file system, which would occupy a large disk space, can be resolved by the existence of a *layered file system*, where Dockers can share parts of their file systems between themselves and with the hypervisor. Their main limitation is basically the labile isolation between containers, which can lead to potential security issues; for this reason, using Dockers in an environment where each container belongs to a different tenant could be not the best solution, because preventing all the possible access attempts becomes difficult in this scenario.

Independently of the technology which is used, an important advantage of the NFV paradigm is the capability of auto-scalability of the *Virtualized Network Functions* (VNFs) every time more resources are needed. Two approaches can be followed in this context:

- *scale-up* when more resources are assigned to the same VNF. The disadvantages of this approach are that the upper-bound limitation is represented by the maximum quantity of computation resources of the server on which the VNF is installed and the implementation of the code (e.g. a mono-thread application) does not always get benefits from receiving more resources for itself;
- *scale-out* when more instances of the same VNF are installed. This approach guarantees parallelization and flexibility, even though it could require a load balancer which considers both the actual load of each function instance and the quintuple of the incoming packets to decide the instance in charge of a specific traffic flow.

Since more Virtualized Networks Functions are needed to provide a network service, orchestration management tools must be exploited. The orchestrators which

can be used are technologies directly belonging to the cloud world, like *VMWare vSphere* or the open-source alternative *OpenStack* for Virtual Machines on one side, *Kubernetes* for Dockers on the other side. Their tasks are to accept commands like the creation of a new service, to decide which functions should be used to provide that service, to choose the adequate implementations, to allocate the needed functions on the servers according to requirements based on computational resources or preferably amount of traffic sent on the network and to monitor the state of the service with functionalities like load sharing between appliances of the same type and self-feeling, i.e. redeploying a function if the previous instance is not running any more.

2.3.2 ETSI NFV Model

The *European Telecommunications Standards Institute* (ETSI), an independent standardization organization, formed together with the leading telecom operators an *Internet Specification Group (ISG)* with the goal to define a set of vendor-independent specifications for the Network Functions Virtualization paradigm.

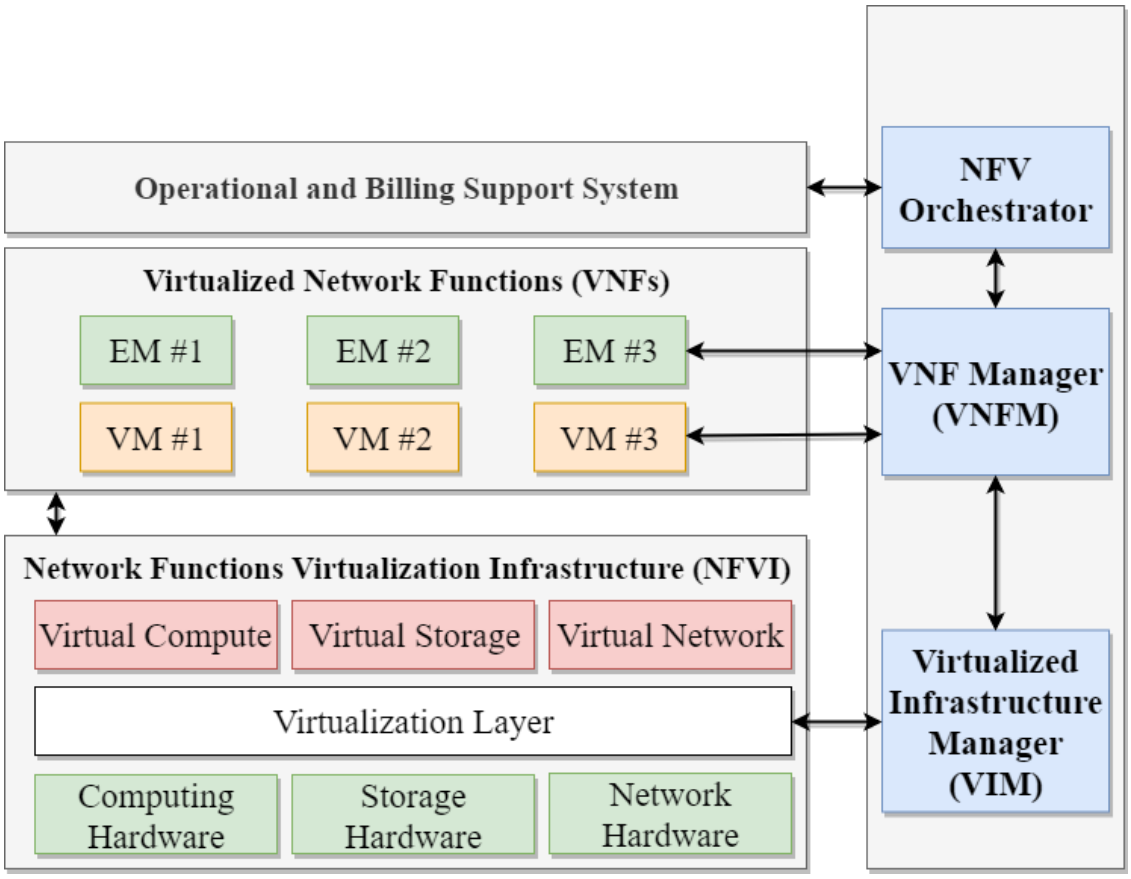


Figure 2.4. ETSI NFV model representation

They established a high-level architectural framework to encompass the management of the network functions, to manage their relationships and the intermediate data flows and to allocate resources accordingly to the needs. These tasks were divided into three different blocks, which are described in the following:

- The *Network Functions Virtualization Infrastructure* (NFVI) block provides both the hardware where the functions can be placed and the software to virtualize them (e.g. the Virtual Machine technology that can be used to deploy the functions). The resources which NFVI offers are the computing hardware including the CPU and memory needed by a function, the storage hardware which can be locally attached or distributed through SAN technologies and the network hardware, characterized by a set of Network Interface Cards the VNFs can use to communicate. Moreover, a fundamental element of the NFVI is the *virtualization layer*, which makes the underlying hardware available as a virtual substrate.

- The *Virtualized Network Functions* (VNFs) block is characterized by the specific software functions which exploit the virtualization software provided by the NFVI block to work without the requirement of dedicated hardware. Each VNF-block is characterized by two interconnected elements, that are the Virtual Machine (VM) on one side, the Element Management (EM) on the other. The VM simply implements the network function which must be provided, whereas the EM implements the management feature to assist the VM during its life-cycle.

- The *Management and Orchestration* (MANO) block manages all the resources in the infrastructure and virtualization layers, through interactions with both the NFVI and VNF blocks in the framework and with a complete overview on the overall network.

Figure 2.4 shows how MANO interacts with the other elements of the framework:

- The *Virtualized Infrastructure Manager* (VIM) manages the computing and storage hardware as well as the virtualization layer in NFVI; it has a complete overview on the available resources and on their operational attributes such as power management, health status, and CPU or memory availability, while monitoring their performance at the same time.

- The *VNF Manager* (VNFM) is responsible of the actual implementation of the virtualized functions; it manages both their life-cycle and the scalability of the assigned resources. In fact, when a new VNF must be instantiated, VNFM communicates its requirements to VIM, because this module has an overview on the current status of the resources by means of a monitoring activity; only after a positive answer, VNFM decides which resources can be dedicated to the new function.

- The *NFV Orchestrator* (NFVO) has the key role to coordinate the complete framework, interacting – if necessary – with an external controller like an SDN Controller or an external application; it overlooks the complete end-to-end service deployment and interacts with both the billing support system and the VIM module in order to have a full overview on how the VNFs are actually deployed on the hardware infrastructure.

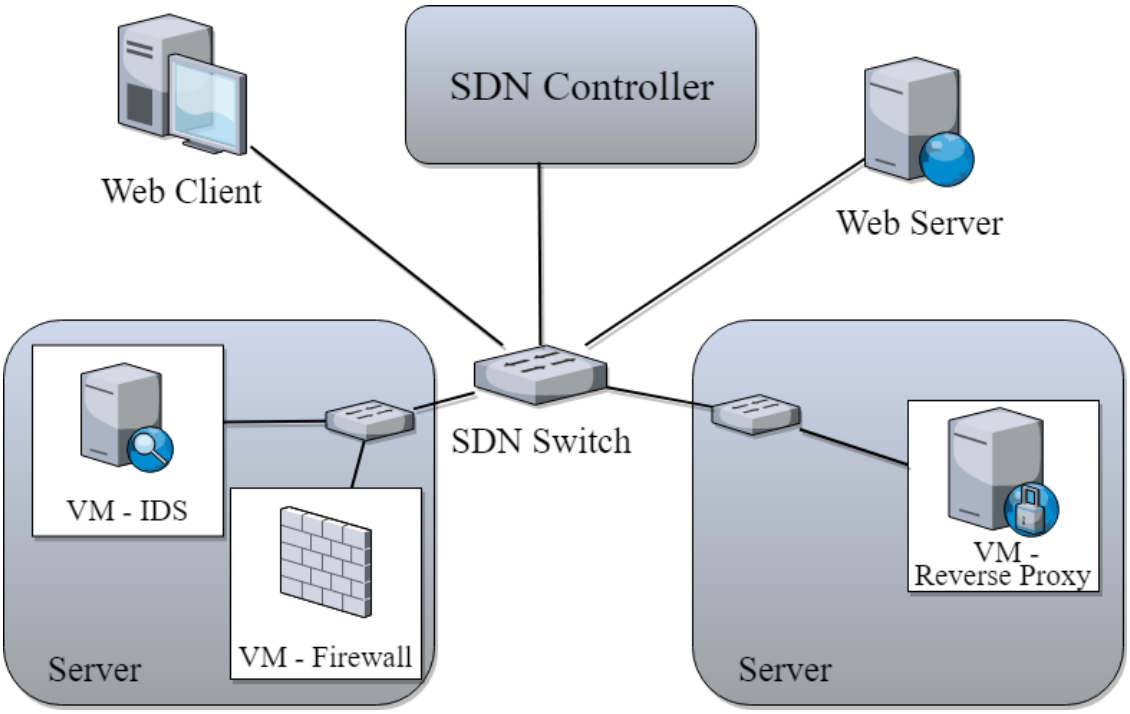


Figure 2.5. Example of Service Function Chain modelled with a NFV-SDN architecture.

2.3.3 Application of the NFV technology to a SFC

Thanks to the Network Functions Virtualization technology, each function does not require to be a specific hardware box in the Service Function Chain, but it can be a software function installed on a server. The Service Function Chain modelled in Figure 2.3 can now be represented as in Figure 2.5, where the functions are not any more specific hardware appliances, but Virtual Machines which can be potentially installed on the same server. The Software-Defined Networks paradigm is used as a complementary tool: the flow of the packets between the elements of the Service Function Chain is determined by the rules which an SDN Controller can lower not only to an external network of SDN switches which put the servers in communication, but also to the software switches which are created in the hypervisor Operating System of each server and which allow the interaction between the deployed functions.

In addition to the benefits which SDN already introduced, the NFV technology applied in relation to a Service Function Chain adds other advantages:

- there is great flexibility by which a new function can be deployed, since it is enough to create a new Virtual Machine or Docker in the Operating System host, instead of purchasing and configuring a hardware box;
- it provides dynamic links for the functions inside the same server thanks to a software switch which can be potentially configured as an SDN switch;
- all the virtualized functions on the same server share the computational resources, which can be allocated dynamically according to the needs;

- it is easy to build a forwarding rule for the software switches in a proactive way instead of a reactive mode, minimizing the interactions between the switches and the SDN Controller. In fact, when a Virtual Machine is created, the MAC address of its virtual Network Interface Card, its position inside the hypervisor and particularly the port number of the software switch to which it is linked are well-known, while in a physical network the point of access of a device cannot be usually predicted in advance.

2.4 Network Automation

These novel paradigms allowed the researchers to put a strong emphasis on the concept of Network Automation, whose purpose is the deployment of a complete end-to-end service by configuring automatically not only the single functions in the edge appointed to provide it, but also the intermediate communication network. The core idea is the flexibility of reactions in changing the configuration both when the user introduces a new requirement and when an event arises from the network itself; in addition, decisions should take care of the current scenario and behaviour of the subjacent physical level.

This approach is becoming fundamental also in the security scenario, where the reaction speed is fundamental to face an increasing variety of cybersecurity attacks which cannot always be predicted in advance. In this case, if a security manager must manually change the security configuration after receiving an alert from functions such as an IDS, in an automatic context the configuration changes could be managed automatically with a slower latency.

For Network Automation, several tools have been developed in the recent years. They allow to interact with the most common implementations of the SDN and NFV principles on one side and to define a higher level of abstraction of the interfaces they provide, in order to be independent from the specific implementation as much as possible.

On the edge side, tools like Chef, Puppet and Ansible can cooperate with the cloud orchestrator, like Openstack, in the deployment of virtualized services on the servers; they provide play-books which are a set of rules corresponding to a determinate function and can be translated in the proper low-level configuration. They can work automatically reacting to the events from the network, not only to the commands, and exploit YAML (*Yet Another Markup Language*) as a human-readable data serialization language for the configuration files.

Instead, on the core side, RESTConf is a management configuration protocol which allows firstly to manage and historicize the configuration of the appliances and, at the same time, create and modify configurations themselves independently from the specific form (e.g. hardware or virtualized appliance) of the function. This is possible by the cooperation with YANG, a data modelling language which can create an abstract data model for each different function and provide a service-agnostic interface through which the input commands are parsified in relation with the data model placed in the configuration.

The goal is to avoid a continuous interaction with the user so that the software can close a loop of reactions by itself, getting information by the network and immediately exploiting them to improve the general behaviour without waiting that a human being would look at the monitoring results to take a decision. On one hand, a problem which can arise is that this automatic process uses configured algorithms to decide which changes should be made to the configuration when triggered by a network event, so it is fundamental that this process has a complete overview on the entire infrastructure, with sensors which allow to monitor the traffic and establish which is the normal behaviour. On the other hand, if this problem is restricted, then the benefit provided by Network Automation is that the system is able to adapt by itself to the different scenarios.

Chapter 3

Policy-based Management and Firewall Auto-Configuration

This chapter introduces the policy-based approach for the management of network security issues, providing the basic terminology of the matter and which limitations of alternative approaches it can overcome.

Then, a brief presentation of network security policy specification is presented with the goal to illustrate the main typologies which are of interest for this thesis. Moreover, the problem of policy refinement is introduced to deal with the necessity of translation from a high-level language to the low-level configuration of specific network functions deployed to satisfy a policy.

Finally, a brief description of the most relevant techniques in literature about firewall auto-configuration in policy-based scenarios is presented. Among these techniques, *Firmato*, a toolkit for an automatic management and configuration of firewalls in a traditional network, occupies a central position, since the proposed ideas have been an inspiration factor for the thesis work.

3.1 Policy-based Management

In distributed systems, security is becoming a fundamental aspect to take in consideration for network functions configuration, because nowadays the *Information and Communications Technology* (ICT) is part of every activity and service. Security properties like data confidentiality and integrity must be enforced to respectively avoid that attackers could have visibility on the information to protect and could modify them.

For these reasons, a burdensome task, which a security manager is in charge of, is to properly configure the network functions implementations which are used to provide a service, taking in consideration not only the traditional issues of efficiency and performance, but also the security aspects. However the variety of different implementations make this task even more difficult, as long as the fact that the creation or configuration of a security protocol are extremely complex and error prone.

A first approach could be that, trying to adopt the attacker mindset, the security manager creates an initial configuration, which will not anyway protect from all the possible attacks since ICT security is, by definition, a reactive process; in fact, even though all the current attack strategies are correctly prevented, it is evident that in the future new ones will be discovered and exploited. Then, every time a misconfiguration is notified or a limitation of the existing configuration is presented, the security manager edits it trying to improve the robustness of the system until the cycle naturally repeats.

To overcome this limitation, in literature a policy-based network management architecture has been proposed. The main idea is to define abstract rules which describe the behaviour of a system and can be used by a security manager – or also a service designer without deep knowledge in this field – to verify that the network functions configuration satisfies the security requirements.

3.1.1 Basic terminology

The RFC 3198 [6] provides formal definitions of the *policy* term and correlated aspects; the most important ones will be presented in the following to clarify the concept of policy-based management.

First of all, a *policy* can be defined from two alternative perspectives:

1. A policy is a definite goal, course or method of action to guide and determine present and future decisions. It is actually fundamental both to provide a formal definition of the expected behaviour – which can be easily referenced – and to perform, potentially in an automatic way, a formal verification of the security requirements for a network.

2. A policy is also a set of rules to administer, manage, and control access to network resources. Each policy can, in fact, refer to a different context (e.g. different policies can refer to different tenants) and then the inner details are formalized with single expressions representing the actual requirements to satisfy.

The creation and enforcement of a policy involve two actors: the *subject*, which is an entity or a collection of entities with the authorization to request the satisfiability of a set of constraints, and the *target*, that is characterized by the elements affected by a policy, such as the network devices or the service functions to allocate and configure.

Then, a *policy rule* is a basic building block of a policy-based system. It is the binding of a set of actions to a set of conditions, where the conditions are evaluated to determine whether the actions should be performed:

- A *policy condition* is a representation of the necessary state and of the prerequisites that define whether a policy rule's actions should be performed. When the policy conditions associated with a policy rule are evaluated to true, then the rule should be enforced, unless other factors such as rule priorities and decision strategies are taken in consideration.

- A *policy action* is instead the formalization of operations which must be performed to enforce a policy rule, when the conditions of the rule itself are satisfied. These operations could affect the flow of the network traffic and the configuration of the network resources.

After the definition of the policy rules, a security manager has an immediate and complete overview of the protection needed by the system and can verify if the requirements are violated by a proposed network configuration. These activities are possible thanks to the matching between the conditions and the actions of a rule, which are correlated and make the functions configuration dependent on the requirements.

In this context, a *rule-based engine* is the tool which can be exploited by a security manager to evaluate the policy conditions and trigger the corresponding policy actions, if necessary. A particular rule-based engine may only be capable of acting upon policy rules that are formatted in a specified way or adhere to a specific language; for this reason, it is fundamental that the policies are specified with a proper syntax.

3.1.2 Policy-based Management Framework

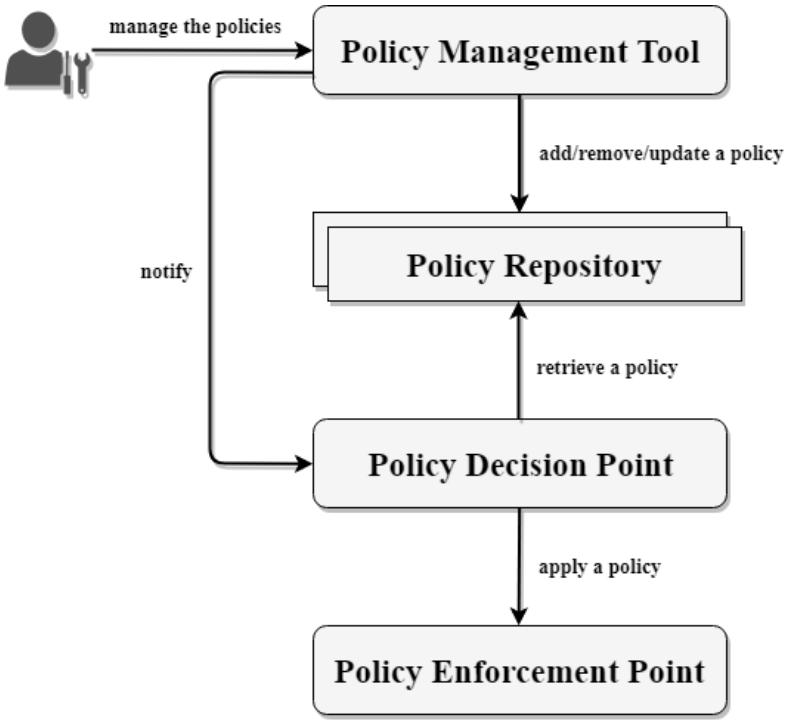


Figure 3.1. IETF Policy-based Management Framework

In the RFC 3060 [7], the *Internet Engineering Task Force* (IETF) proposed a basic architecture for the policy specification and management, which has been then further extended in literature with more capabilities specific to the interested areas.

The main elements, whose interactions are shown in Figure 3.1, are the following four:

- the *Policy Management Tool* (PMT) allows the user to specify the network security policies which must be enforced and translates them in the form of single rules or combination of rules, each one of which is characterized by a set of conditions and a set of actions and is represented in a lower language than the user's specifications;
- the *Policy Repository* (PR) is a database of the policies which the Policy Management Tool created;
- the *Policy Decision Point* (PDP), every time it is notified by the Policy Management Tool after any modifications of the rules in the Policy Repository, is in charge of reading the database and taking decisions based on the requirements;
- the *Policy Enforcement Point* (PEP) is the module where the decisions taken by the Policy Decision Point are finally applied and the policy is actually enforced.

As it can be inferred by the flowchart, most of the operations are automatically fulfilled by the framework and the only interaction with the security manager is in the phase of the high-level policies specification. If on one side it can avoid the mentioned problems about an approach based on notifications of configuration mistakes, on the other side the complexity is here moved to the policy specification.

If the policies are wrongly specified or they are in conflict, the resulting configuration computed by the Policy Decision Point can be exposed to potential security attacks or can block traffic which should be allowed. It is, consequently, fundamental to define a high-level language that the users can exploit to express the network policies, a translation module of these expressions in lower level rules and a conflict analysis process to detect conflict anomalies in the requirements.

3.1.3 Policy specification and abstraction

In the specification of the policies, three main levels can be individuated:

- *high-level policies*, which must allow to express all the requirements in a user-friendly language. Considering an *action-object-attribute* language, examples of possible high-level policies expressions are "block all the traffic to social networks" and "log access to all the websites".
- *medium-level policies*, which are translated from high-level policies and allow the creation of an intermediate format, independent from the specificity of the network functions implementation which will be used to provide the service. For example, a high-level policy such as "allow all the web traffic" can be translated in a rule which allows all the packets having as source or destination TCP port the number 80, independently of the function on which it will be enforced;

- *low-level policies*, corresponding to the proper configuration of the selected network functions and potentially translated from a set of medium-level policies, because each function can include more single capabilities. They are consequently dependent on the chosen function implementation.

In literature several languages have been proposed for the policy representations: IETF proposed the *Policy Core Information Model* (PCIM) [7] to work with the infrastructure model defined in RFC 3060, while the *Organization for the Advancement of Structured Information Standards* (OASIS) defined the XML-based language *eXtensible Access Control Markup Language* (XACML) [8] for the representation of Access Control Policies.

This thesis followed the approach presented by Valenza and Lioy [9], who defined two user-oriented network security policy languages, called *High Security Policy Language* (HSPL) and *Medium Security Policy Language* (MSPL); the main purpose is to provide an abstraction from the proper low-level configurations of the functions, which must strictly follow the technical specificities of the corresponding documentation and that for this reason cannot be adopted for the upper levels of the aforementioned policy hierarchy.

HSPL is a language used for high-level policy specification and abstraction, based on a *subject-verb-object-parameters* paradigm; each expression is, in fact, characterized by a *subject* that represents who wants to enforce this network security requirement, a *verb* used to indicate the action which must be performed, an *object* as the target of the action and finally additional *parameters* to provide more richness and completeness.
The main HSPL characteristic is that it must allow the creation of simple expressions, which can be understood and formulated by end users without specific and detailed knowledge; a possible auxiliary tool could be a predefined list of the values for each one of the rule fields. In addition, this language must be extensible and flexible, so that it provides ease in the introduction of new kind of policies and in the support of a variety of conditions to respect and actions to perform.

On the other side, MSPL is a language which not only requires extensibility and flexibility, but it must be able to provide all the information which in the next step are required for the network functions configuration; for this reason, its targets are technical people, like the security manager. Anyway, the form in which the MSPL rules are expressed must be abstract and independent of their specificity in order not to be linked to specific implementations.

3.1.4 Policy refinement and translation

In the hierarchy presented in Subsection 3.1.3, two fundamental processes which link the different policy typologies, as showed in Figure 3.2, are:

- *policy refinement* for the transformation of HSPL expressions in MSPL rules;
- *policy translation* for the transformation of MSPL rules in the low-level configuration of the network functions.

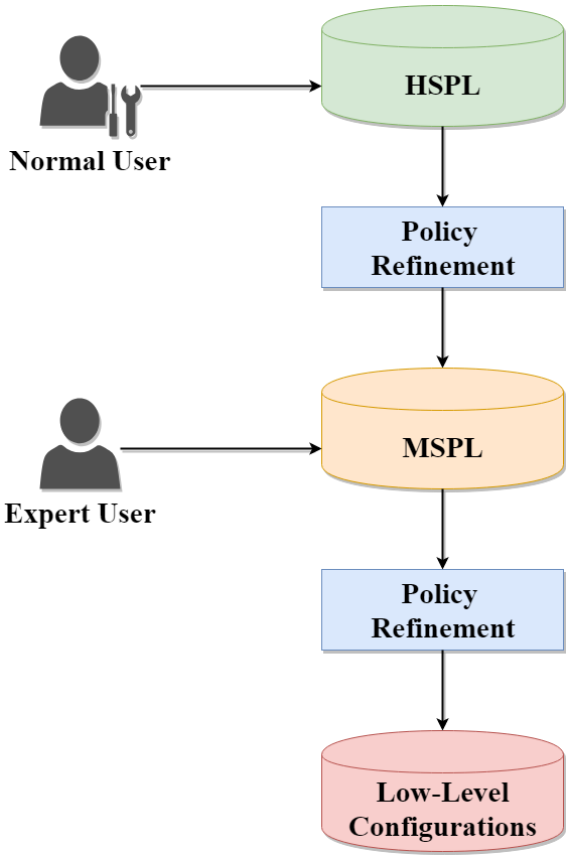


Figure 3.2. Model of the policy refinement and translation operations

Policy refinement covers a fundamental role because it is the transformation process which allows the creation of the medium-level policies from the high-level requirements specified by the end user; furthermore, this process is prone to automation. The main purposes of policy refinement are, in fact, the following ones, as they have been identified by Moffett and Sloman [10]:

1. translation from high-level to lower-level policies without losing the information carried by the specified requirements;
2. individuation of the resources which are required for the satisfaction of the policies;
3. verification that the determined low-level policies fulfil the high-level ones and that the individuated resources are sufficient to properly satisfy them.

Among the refinement techniques which have been proposed in literature, Zhou and Alves-Foss [11] presented a policy refinement method for *Multi-Level Security* systems based on architectural design patterns. Their idea is to represent the requirements by means of policies defined by the triple *p=(Intra, Inter, Domain)*, where *Intra* is the set of internal rules related to the specific policy, *Inter* is the set of the rules which model the relationships with other policies and *Domain* is a set of associated services. Through architectural patterns like Decomposition Pattern,

Aggregation Pattern or Elimination Pattern, it is possible to model abstract components in a set of refined components of the policies, which do not require to be the same number as the original ones; then, for each part of the policy, additional rules are modelled to express how it is linked to the others.

A different goal-based approach was proposed by Bandara et al. [12]. Since the high-level policies express the behaviour the system should have to satisfy some goals, this technique tries to connect these goals not only between themselves, but in the refinement phase also to the specific elements of the system; the purpose is to establish a complete strategy to reach the goals which have been defined by the end user. It is a powerful approach because it allows to provide a complete abstraction of the HSPL rules.

Instead, the *policy translation* is the process which transforms medium-level security policies into the low-level configuration of the specific network functions. Since MSPL expressions must already contain all the needed information, this phase is trivial because it only requires a syntax change between abstract and implementation-independent rules and the specific language of the used implementation.

3.2 Firewall Auto-Configuration

A firewall is an essential element in the design of a network security architecture, whose main purpose is to block all the dangerous and unwanted traffic incoming from an external network, where the security level is inferior because it is not possible to establish who is able to make a connection, to an internal network or *intranet*, where a more accurate control on the accesses is provided and monitoring systems like Intrusion Detection Systems (IDS) are installed.

The critical aspect of the firewall management is to correctly configure its Filtering Policy, that is the set of rules by means of which a firewall decides if a received packet must be discarded or forwarded to the out-ports. If this task is performed manually, not only it requires a not negligible time for a security manager because he must consider several security requirements to be satisfied, but at the same time it is prone to human errors such as conflicts or anomalies.

For the aforementioned reasons, the firewall Filtering Policy configuration can be used as an example to introduce the concept of *Security Automation*, a novel paradigm which exploits automatic control processes to establish the policies of the security functionalities in an automatic way, considering a set of requirements or events coming from the network itself. In this section, the most relevant works in literature about firewall auto-configuration are presented, with a brief description of the introduced novelties and their drawbacks; all these studies have been taken in consideration during this thesis work.

3.2.1 Firmato: a firewall management toolkit

A firewall is usually installed on gateways, like routers or level-2 bridges, and it is rule-based, because a set of *match* - *action* rules are configured in order that, if a

packet satisfies the requirements provided by a rule, then the relative actions must be performed on this packet.

However, the typical approach of a manual configuration performed by the security manager has some considerable drawbacks, as Bertal et al. [2] detailed:

- the probability of a manual misconfiguration is high because the rules are applied in a specific order to the incoming packets and there is not any kind of automatic control on potential redundancies of the installed rules inside the firewall;
- the security of the internal network depends on the precise configuration of these rules, without providing any form of higher abstraction levels;
- if more firewalls are present in the network, their configuration is more problematic because they do not represent a unique repository of rules, which would be easier to manage and debug.

To overcome these limitations, *Firmato* is a toolkit proposed to automatically configure traditional firewalls by exploiting network security policies expressed with a *Model Definition Language* (MDL), which allows the creation of *roles* representing network permissions for a group of hosts. In this architecture, the *Model Compiler* performs a translation of the model created with this medium-level language in the configuration files of the specific firewalls, distributing the rules belonging to a single model to multiple instances of this network function. The abstraction levels which Firmato introduced were on one hand a separation between the policy design and the vendor specificities of the firewalls, on the other hand a separation between the policy rules definition and the network topology.

The aspects characterizing *Firmato* toolkit influenced the design of the ADP module of VEREFOO, later described in Section 5.3, but its application is related to traditional networks, where novel technologies like NFV and SDN are not exploited.

3.2.2 Other works about firewall auto-configuration

In the following years, other research works focused on improving the methodology by means of which firewalls can be configured not only in traditional networks, but also in the NFV and cloud environment; most of them identify novel techniques which correct or create the firewall policy, identifying the needed rules to satisfy a set of security constraints.

Youssef and Bouhoula [13] proposed a fully automatic approach for fixing firewall misconfiguration. The framework they developed, based on a *Soundness Completeness Verification*, is both a correction tool because it is able to identify which rules in a firewall policy should be corrected to remove the presence of some conflicts, but it is also an optimization tool. Actually, exploiting Yices [14] as an SMT solver, its main purpose is to automatically reduce the number of rules in the policies and establish inference rules that permit to detect and delete the shadowed rules. The fundamental aspect which can be extracted by this work is the

optimization-oriented approach, but it is still applied to a traditional scenario where firewalls are hardware middleboxes.

Gember-Jacobson et al. [15] then developed *Control Plane Repair* (CPR), a framework which automatically repairs network control planes by exploiting an abstract representation; it is clearly a forward step in the direction where the functions which compose a Service Graph are independent from the actual implementations deployed on the physical infrastructure and at the same time it offers an automatic methodology to identify misconfiguration. Moreover, this approach is based on a MaxSMT problem to minimize the number of routers and links to update when the service configuration must be changed. On the other hand, the main limitation is that it focuses on identifying problems in an existing security configuration, instead of creating it from a set of security requirements.

Recently, Adi et al. [16] exploited process calculus to verify if the configuration of network functions, including firewalls, respects a set of security constraints; in negative case, an automatic modification of their configuration is performed by means of an enforcement process that imposes the behaviour of the system's model as stated by the input requirements. Like in the previous case, the main limitations are that it only automatically modifies existing wrong configurations and does not perform any automatic allocation operation of the functions in the Service Graph.

Chapter 4

Tools: z3 and Verigraph

In this chapter the two most important tools for this thesis work are presented, with a short description of their main features and the goals for which they were needed. In particular, the structure of the chapter is the following:

- in Section 4.1, the z3 theorem prover is described, with a focus on how it can model a MaxSMT problem, which represents a fundamental aspect of the thesis;
- in Section 4.2, the Verigraph tool is illustrated, with a focus on the data structures and functions that have been used during the development phase of the thesis.

4.1 z3

4.1.1 Introduction to z3

z3 [17] is a state-of-the-art theorem prover developed by Microsoft Research which can be used in software analysis and in the verification context to solve all the *Satisfiability Modulo Theories* (SMT) problems, representing a generalization of the *Boolean Satisfiability* (SAT) problem by the introduction of additional theories, like equality reasoning, arithmetic, fixed-size bit-vector and quantifiers.

It offers APIs (Application Programming Interfaces) in different high-level programming languages, like python, Java, C and C++. In this thesis work, Java APIs in the version 4.8.1 for 64-bit machines were used. Appendix A provides a guide about how using the Java z3 APIs exploited in the development phase.

Figure 4.1 describes how the z3 framework works. After receiving a set of formulas through a programming interface, it translates them in a SMTLIB2 file, according to the language and semantics described by the corresponding international initiative to provide a common background for SMT theories. Then, it firstly tries to apply tactics like pre-processing or heuristics, respectively to limit the total computation time or to get a suboptimal solution. Finally, it exploits a specific solver (e.g. SAT, Fixedpoint) to get a solution or, in case an optimization phase is

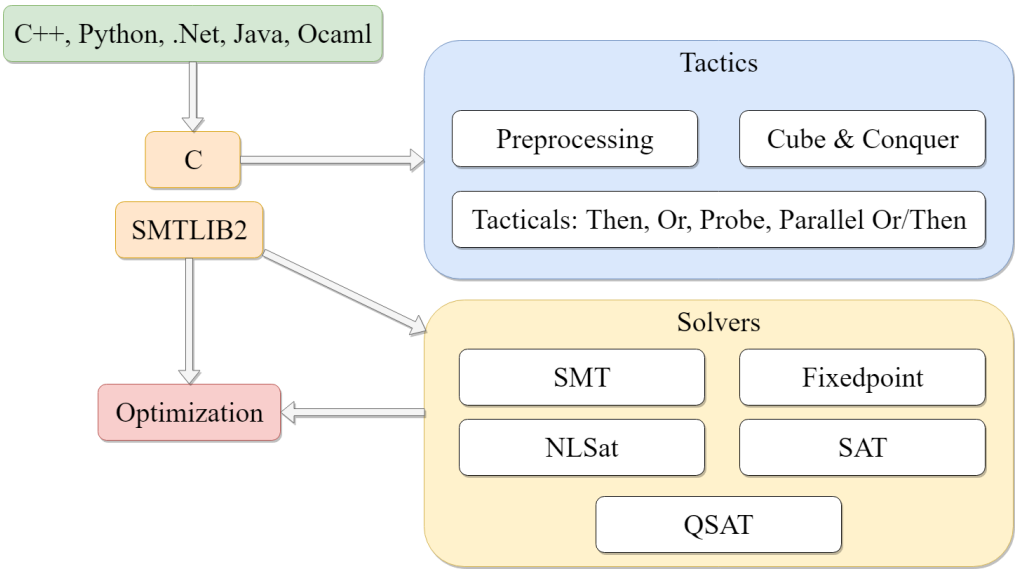


Figure 4.1. z3 architectural model

needed, to get the optimal solution; for the optimization purpose, in particular, z3 exploits an optimizer engine, called *z3Opt*.

4.1.2 The SMT problem

The SAT problem, also called the propositional satisfiability problem, is a traditional problem in the computer science field whose goal is to determine, given a propositional formula with a set of propositional boolean variables (i.e. that can assume as possible values only true or false), if a combination of values for these variables exists to guarantee the satisfiability of the formulas. A SAT solver, for this reason, does not require to find the best combination of the propositional variables, but a single solution among all the possible ones is sufficient.

Nevertheless, the SAT problem is *NP-complete*. It belongs to NP class, where NP means "non-deterministic polynomial time", because it can be solved in polynomial time using a non-deterministic Turing machine; in other words, the correctness of the formulas expressed in a NP problem can be evaluated rapidly in a polynomial time, but the time to find a proper solution increases non polynomially with the size of the problem itself. Besides, it is NP-complete because every other NP problem can be reduced to the SAT problem in polynomial time.

The language which is exploited by the SAT solvers is the traditional boolean logic. A generalization is represented by the *Satisfiability Modulo Theories* (SMT) problem, where the language is instead the first-order logic, which includes the boolean operations as a specific case, but it can use several other theories, such as theories of real numbers, integers, lists, arrays, bit vectors and many other data structures. Basically, a SMT problem is composed by a set of predicates, where each predicate is a binary function defined over non-binary variables. Consequently, the SMT language is much richer than the SAT language, it allows to express more

complex models and it represents a formalized approach to constraint programming for constraint satisfaction problems.

To solve SMT problems, z3 can exploit a specific SMT solver integrating search pruning methods, but also heuristic combinations of algorithmic proof methods called tactics; they are characterized by several parameters whose tuning is needed to get good performance results.

4.1.3 The MaxSMT problem

The *Maximum Satisfiability Modulo Theories* (MaxSMT) problem is an extension of the SMT problem in the optimization context, where given a set of predicate clauses containing predicate variables, the goal is to find the optimal values of these variables which achieve the maximum satisfiability of the clauses.

Like the SMT problem, the MaxSMT problem is NP-*complete* in terms of worst-case computational complexity. The main difference is, however, that each clause is assigned a weight, unitary in the standard version; consequently, it is not sufficient to find a solution which satisfies the predicate clauses, but among all the possible solutions the chosen one must maximize the number of satisfied clauses.

The most common MaxSMT variants are:

- the *weighted MaxSMT*, where a different weight can be assigned to each clause and consequently the research for the best solution prioritizes the satisfiability of the most valued clauses;
- the *partial MaxSMT*, where some constraints are not relaxable because they must be satisfied, while other clauses do not require to be necessarily satisfied for the achievement of a solution;
- the *weighted partial MaxSMT*, which is a combination of the two previous instances.

Considering a weighted partial MaxSMT problem characterized by S relaxable clauses s_i, also called *soft constraints*, and H clauses h_j which must be satisfied, called *hard constraints*, a formal definition of the problem can be the following:

$$max \sum_{i=1}^{s} w_i * s_i$$
$$subject\ to\ h_j, \forall j \in [1, H]$$

z3 language offers support for modelling both kinds of constraints:

- *assert* declares a hard constraint;
- *assert-soft* declares a soft constraint.

Listing 4.1 shows an example of weighted partial MaxSMT modelled in z3 language, while Listing 4.2 shows the corresponding solution:

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)
(declare-const d Bool)
(declare-const e Bool)
(assert-soft a :weight 3 :id A)
(assert-soft b :weight 2 :id A)
(assert-soft c :weight 1 :id A)
(assert-soft d :weight 2 :id A)
(assert-soft e :weight 3 :id A)
(assert (not (and b d)))
(assert (not (or e b)))
(assert (not (and a c)))
(check-sat)
(get-model)
(get-objectives)
```

Listing 4.1. MaxSMT problem in z3 language

```
sat
(model
(define-fun c () Bool
false)
(define-fun a () Bool
true)
(define-fun d () Bool
true)
(define-fun b () Bool
false)
(define-fun e () Bool
false)
)
(objectives
(A 6)
)
```

Listing 4.2. MaxSMT solution in z3 language

4.2 Verigraph

4.2.1 Introduction to Verigraph

Verigraph [18] is a tool developed by the Netgroup working group at Politecnico di Torino to verify the satisfiability of network policies within network graphs. The scenario in which Verigraph was developed is the *Service Provider DevOps* (SP-DevOp) context, where the four principles of DevOps methodologies which were

originally designed to work in a data center environment are applied to a Service Provider infrastructure [19]:

Observability The first phase of the process monitors the state of the network and analyses the flows of the packets.

Trobleshooting After a preliminary analysis, the operation quality is evaluated to identify potential risks.

Verification Before creating or changing the configuration of some network functions to solve the previously identified risks, automatic tools must perform a formal verification of the satisfiability of the policies in the network in which they must be enforced.

Development The new configuration of the network functions is developed and deployed.

These four phases represent a development cycle, because when a new configuration is deployed or a new network function in installed, then a sequent monitoring phase is needed to understand the new behaviour of the network.

In this context, Verigraph can perform a Service Graph verification in an NFV and cloud context; Section 5.1 will define the concept of Service Graph with more accuracy, but it basically represents the logical topology of a network where each node is characterized by a specific network function, such as a traffic monitor or a web cache.

4.2.2 Verigraph Network Model

Verigraph exploits z3 to define a formal model of a network, by means of data structures and methods which represent the fundaments of the *ADP* module of VEREFOO, described in Section 5.3. This subsection briefly describes the most relevant aspects of the model and provides useful terminology which will be recurrent in this thesis.

A *packet* which a node of the network can receive, process and send is modelled with the following features:

src it is the IP address of the node which sends the packet to reach a specific destination in the network;

dst it is the IP address of the node to which the packet is sent by a specific source in the network;

inner_src it is the source IP address of the encapsulated packet, if an encapsulation operation has been performed;

inner_dest it is the destination IP address of the encapsulated packet, if an encapsulation operation has been performed;

origin it is the name or the IP address of the node which created the IP payload and it can be different from the source IP address if the original packet has been modified by middleboxes such as a NAT;

origin_body it is the original body of the packet created by the source, without any alteration;

body it is the body of the packet, sent by a previous hop to the next hop in the path from the source and the destination, with possible modifications;

seq it is the sequence number of the packet inside a level 4 flow;

proto it is the protocol type of the packet (the supported protocols are HTTP, POP3 and SMTP);

emailFrom it is the email address of the sender, in case the protocol type is POP3 or SMTP;

url it is the URL of the resource obtained by means of HTTP;

options it is a field which describes the most relevant options of the packet headers;

encrypted it is a flag which notifies if the body is encrypted or the packet is encapsulated in another.

The methods offered by Verigraph to model the *flow* of the packets inside the network are the sending and receiving actions, expressions which can be further used in the definition of soft and hard constraints in a MaxSMT problem:

send(n1, n2, pk0) this method creates the z3 expression representing the event of sending the packet *pk0* from the node *n1* to node *n2*;

recv(n1, n2, pk0) this method creates the z3 expression representing the event of receiving a packet *pk0*, sent by the node *n1* to node *n2*.

By means of the packet data structure and the two methods above described, it is possible to define a complete formal model of a loop-free network, where a client can interact with a server through a graph, from which all the possible chains are extracted to establish if the source of a packet can reach its target destination.

During the flow of the packet through the graph modelled with this tool, it can pass through a set of middleboxes. This term was coined by Lixia Zhang and the RFC 3234 [20] defines a middlebox as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host".

The most relevant middleboxes which are modelled in Verigraph are the following:

- *anti-spam*, which drops packets with a POP3 message containing an URL from a blacklisted mail client or server;

- *Network Address Translator* (NAT), with a stateful behaviour which allows the propagation of a packet to an external address if the internal IP address has been previously registered in the module and allows packets directed to internal addresses if they belong from external addresses;
- *web cache*, which can store the contents of web servers to provide them to web clients by means of HTTP messages;
- *ACL firewall* (blacklisting packet filter), which drops a received packet if the source, the destination or the combination of the two addresses are present in a blacklist;
- *field modifier*, which can modify the fields of the received packets before forwarding them;
- *Intrusion Detection System* (IDS), which can monitor the presence of specified words inside the received packets;
- *VPN access* and *VPN exit*, which represent the end points of a VPN tunnel where the original packets are encapsulated in external packets whose source and destination addresses are those of these two middleboxes.

The developed framework, VEREFOO, which will be described in Chapter 5, inherited the verification model from Verigraph and used it as foundation to provide the allocation of the Network Security Functions in a logical topology, their auto-configuration to satisfy Network Security Requirements and the placement of the corresponding VNFs in the substrate network.

Chapter 5

VEREFOO Model

5.1 Introduction to VEREFOO

VEREFOO (*VErified REfinement and Optimized Orchestration*) is a framework whose main purposes are a refinement of the high-level *Network Security Requirements* (NSRs), an optimal allocation and automatic configuration of the selected *Network Security Functions* (NSFs) to satisfy the security constraints and the placement of each implementation of all the virtual functions of the Service Graph on the servers of a physical substrate network. It exploits the *z3Opt* engine for solving the *Maximum Satisfiability Modulo Theories* (MaxSMT) problem in a cloud context, in which it works closely with an NFV orchestrator for the deployment of the correct implementation of the selected Network Security Functions.

For a better comprehension of the general description of the complete framework presented in this chapter, in this section a glossary of useful and recurrent terms is provided:

Service Graph It is a logical topology characterized by a set of network functions, which are used by a service designer, i.e. the person in charge of the service definition, to compose a complete end-to-end service; some examples are the web cache, the load balancer and the traffic monitor. It is basically a generalization of the concept of Service Function Chain presented in Section 2.1: in fact, if an SFC is a chain of network functions, a Service Graph is a more complex topology, where multiple paths can exist between every pair of end points and some loops can be present. The model of a Service Graph requires, consequently, more constraints to take in consideration. In the context of VEREFOO, nevertheless, the creation of a Service Graph does not require to consider any security requirements, so it does not include Network Security Functions such as firewalls and anti-spam filters.

Allocation Graph It is a logical topology, internally created from a Service Graph, which is characterized by the same set of network functions of the Service Graph itself, but also by additional nodes. These extra elements are placeholders called *Allocation Places*, where a Network Security Function could be allocated if this is its optimal position. More generally, if at least one

placeholder exists, in this thesis work a graph is called Allocation Graph, because the decision to allocate a specific Network Security Function on it has not already be taken. The configuration of the Network Security Functions allocated in the Allocation Graph is automatically provided, while the other network functions or the nodes which behave as simple forwarders to propagate the packets in all the possible paths require their configuration to be defined by the service designer.
Both the Service and the Allocation Graphs do not necessarily correspond to the topology of the physical infrastructure by means of *network slicing*, a key principle of *Software-Defined Networks*. A more detailed description, alongside with details about their implementations, is provided in Section 6.2.

Physical Graph It is the physical infrastructure on which the specific implementations of the network functions of the original Service Graph and the Network Security Functions allocated in the Allocation Graph must be placed, with a proper low-level configuration. It is composed by a set of servers which interconnect the end points and are characterized by parameters like the CPU usage, RAM availability, number of deployable functions; these parameters are used to establish, if possible, the best placement schema of the functions on the servers. It clearly exploits the concept of Network Functions Virtualization, presented in Section 2.3.

5.2 Model description

Figure 5.1 presents a complete overview of the framework. In this section, a brief description of each module is provided with the purpose to give a general idea of the workflow; then, in the next sections, more details are showed about the framework element on which this thesis work focused.

The service designer who uses the framework can introduce as input:

- a set of Network Security Requirements to express the security constraints which must be satisfied, by exploiting a high-level or a medium-level language depending on the experience level of the user, through a *Policy GUI* which must make the creation of the requirements easier;
- a Service Graph or, in alternative, directly an Allocation Graph – if he feels confident about defining at priori the positions of the Allocation Places where the Network Security Functions can be installed – through a *Service GUI*, which provides access to a *Network Functions Catalogue* from which the user can decide which functions – simple network functions or also Network Security Functions – immediately allocate on his graph.

A preliminary phase is represented by the *Policy ANalysis* (PAN); the goal of this module, which receives in input the Network Security Requirements, is to perform a conflict analysis, to determine 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

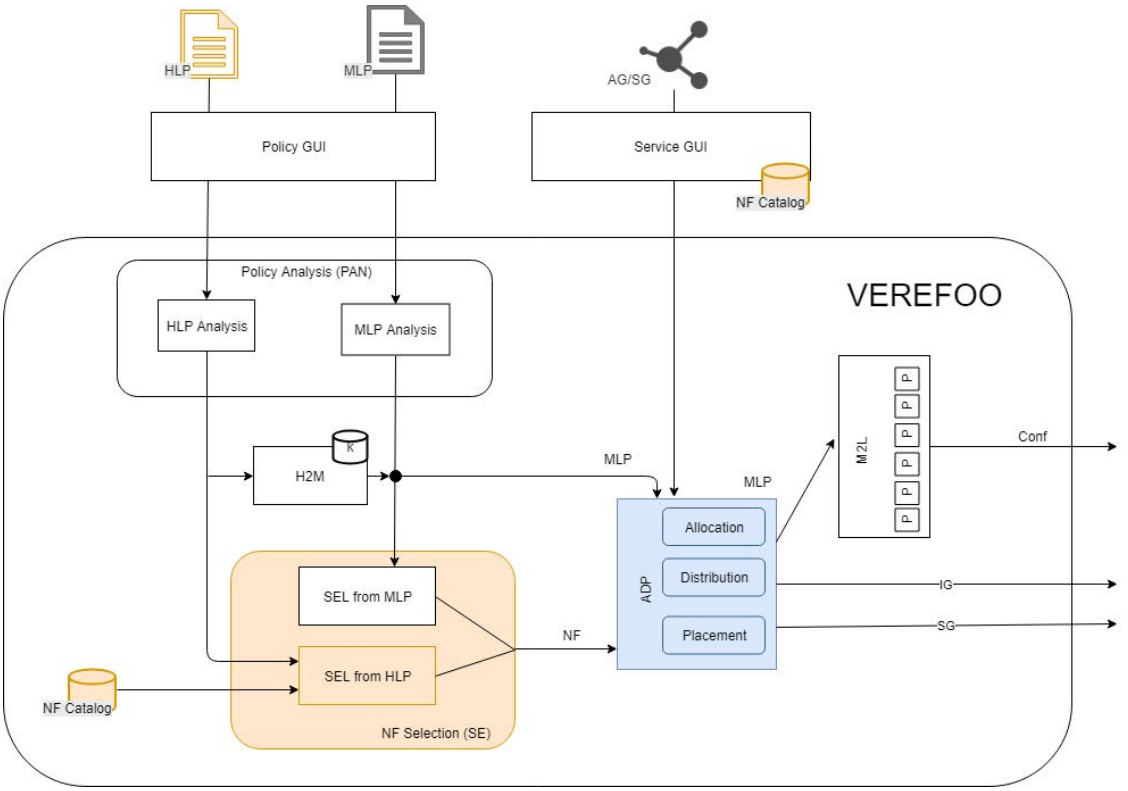


Figure 5.1. VEREFOO model

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 Network Security Requirements, which contain all the useful information for the future creation of the policies of the Network Security Functions automatically allocated on the graph and the low-level configuration of the VNFs placed on the substrate network.

Then, a key role is covered by the *NF Selection* (SE) module; based on the input high-level and medium-level Network Security Requirements, it decides which Network Security Functions are required to satisfy them, choosing them from a pre-built catalogue, which is the same list the service designer has access through the *Service GUI*. This step can require an optimization process by means of which the optimal set of Network Security Functions is selected, even though it does not have any knowledge about the topology of the Allocation Graph and this aspect could represent a possible limitation.

The *Allocation, Distribution and Placement* (ADP) module is the central element of the architecture, whose purpose is to compute a Service Graph with the added Network Security Functions or a Physical Graph receiving as input the medium-level Network Security Requirements, the list of selected Network Security Functions and the original Service Graph or directly the Allocation Graph; more details are provided in the next sections, because it is the module for which this thesis provided a contribution in the development.

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-Level* (M2L) module, which performs a translation of the vendor-independent expressions into the specific rules which must be set on the proper function.

5.3 Allocation, Distribution and Placement

The ADP module of VEREFOO uses *z3Opt* as a MaxSMT solver and Verigraph as a tool for Network Security Requirements verification to provide three main features:

1. given a list of Network Security Functions 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 Network Security Requirements expressed by means of the medium-level language;

2. in contemporary with the allocation phase, a second task is the distribution of the policy rules on the allocated Network Security Functions, always expressed in medium-level language but not necessarily identical to the input Network Security Requirements formulation, because the requirements can be minimized according to optimization goals;

3. in a secondary step, after the creation of the Service Graph enriched with the Network Security Functions, the VNFs implementing the network functions of the original Service Graph and the added Network Security Functions are placed in the physical infrastructure following the principle of minimizing the resource consumption and at the same time the policy rules of the Network Security Functions are translated in the low-level configuration of the VNFs themselves.

An important aspect is that these tasks must be performed with some optimization goals, in order to reach the purpose of the framework of an optimized orchestration in NFV environment:

- minimizing the number of allocated Network Security Functions, so that the corresponding number of VNFs to deploy on the substrate network is inferior, with evident benefits for the resource consumption independently of the deployment strategies which are adopted;

- minimizing the number of policy rules configured for each allocated Network Security Function to satisfy the input Network Security Requirements (e.g. if a group of input requirements express the need of isolation between a group of end points belonging to the same IP subnetwork and a server, these constraints can be satisfied by a single policy rule of a packet filter by means of a

CIDR address range and, in the framework, of the *wildcards* tool, which will be explained in more details in Section 7.4). This step is fundamental not only to save memory for storing the policy rules, but particularly to improve the efficiency of the security operations; in the specific case of a packet filter firewalls, is the set of configured rules is shorter, the decision to forward or drop any packet will require less time.

These goals are expressed in form of soft constraints of a weighted partial MaxSMT problem, previously described in the Section 4.1.3, because the optimal result would be to use the minimum amount of resources that are needed for the deployment of the VNFs on the network and for their configuration; at the same time, the medium-level Network Security Requirements and the topology of the input Service Graph or Allocation Graph are modelled by means of hard constraints, which cannot be relaxable and require satisfiability to let the solver achieve a solution. However, for sake of conciseness, in the remainder of this thesis the weighted partial MaxSMT problem will be simply referred to as a MaxSMT problem.

5.4 Scenarios

The usage scenarios of the *ADP* element of VEREFOO can be divided in two different tasks, which can be performed independently or combined, depending on the user's needs:

- *AOC* (Automatic Orchestration and Configuration), if the objective is to allocate in the placeholders of the Allocation Graph and configure the selected Network Security Functions, identified in a previous step of the overall process, to satisfy the input set of medium-level Network Security Requirements;
- *AVP* (Automatic VNFs Placement), if the objective is to place the VNFs implementing all the network functions of the output Service Graph of the *AOC* task on the Physical Graph representing the underlying substrate infrastructure characterized by a set of servers between the end points of the network.

5.4.1 Automatic Orchestration and Configuration

In the first scenario of the *Automatic Orchestration and Configuration* task, illustrated in Figure 5.2, the initial security configuration of the network should be achieved. For this purpose, the *ADP* module can receive as input:

1. the list of medium-level Network Security Requirements which must be satisfied and which could come from the high-level user specifications;
2. the list of the selected Network Security Functions to satisfy the corresponding Network Security Requirements;

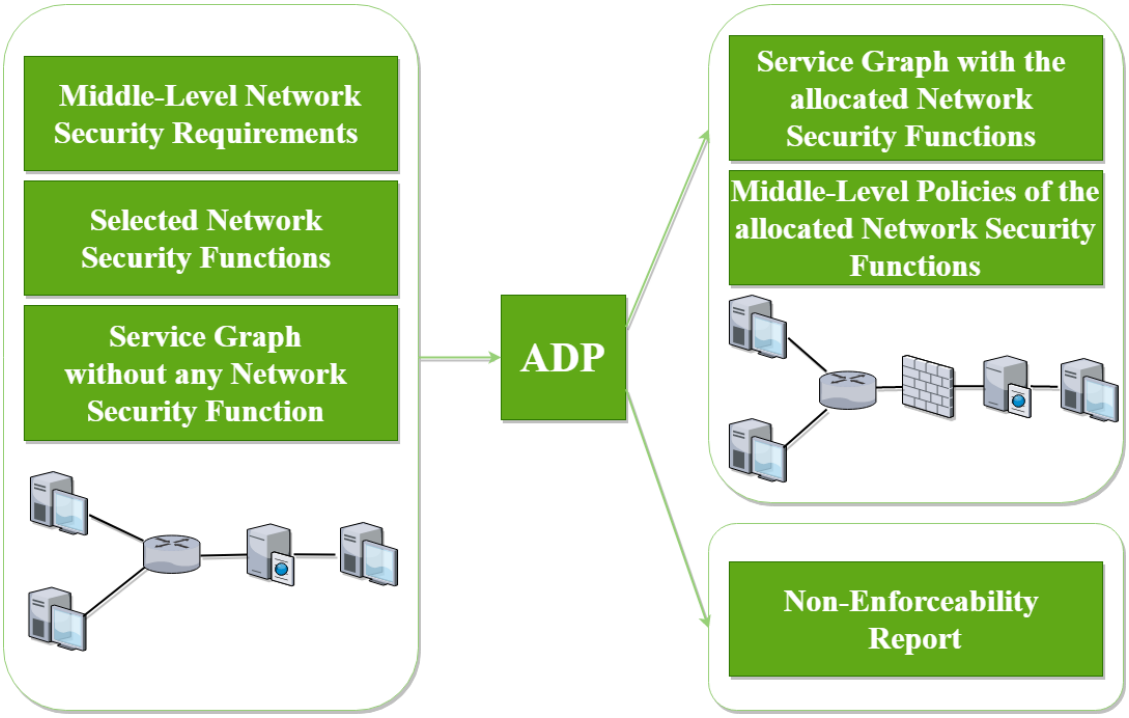


Figure 5.2. First scenario of *AOC*

3. the Service Graph, i.e. the logical topology made by the network functions which are included to offer an end-to-end complete service, with some possible indications about the positions where the Network Security Functions must or must not be placed, if the service designer has some sufficient security knowledge to take these decisions.

In this situation the output could be:

- in case of success, a Service Graph enriched by the Network Security Functions which have been optimally allocated in the Allocation Graph automatically generated from the input logical topology and whose configuration has been automatically created and presented to the service designer by means of medium-level policy rules to satisfy the input Network Security Requirements;
- in case of failure, a non-enforceability report to notify the service designer about the impossibility to compute the Service Graph and to underline which problems occurred during the execution of the module in charge of the MaxSMT problem instance. The main reasons for which in this scenario the *ADP* element could not succeed in computing the result could be that the selected Network Security Functions are not sufficient to satisfy all the Network Security Requirements, the placeholders of the Allocation Graph are not sufficient to allocate all the needed Network Security Functions or a proper configuration cannot be computed because of the topology of the input Service Graph.

This first scenario is a fundamental step when a new complete network security configuration must be automatically computed; however it is not always the most common scenario, since a partial configuration has often already been achieved.

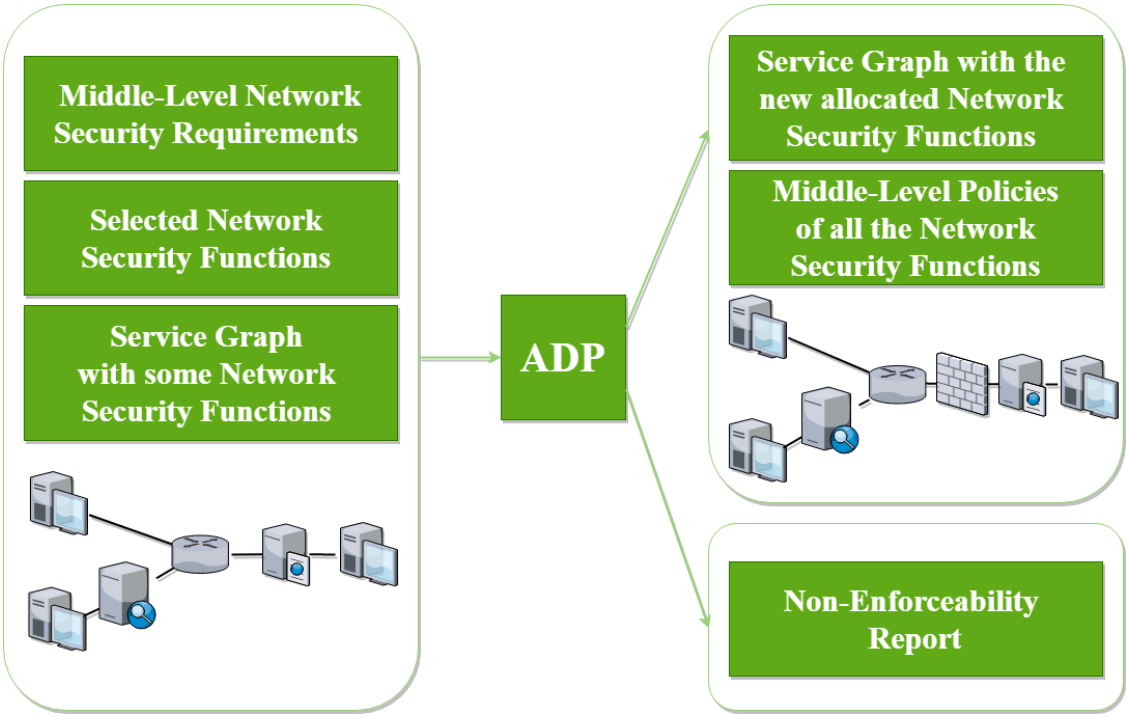


Figure 5.3. Second scenario of *AOC*

For this reason, another typical usage scenario of the *ADP* element is the scenario illustrated in Figure 5.3, which usually occurs in an intermediate phase of the network management after the computation of a previous Service Graph. The goal is to compute a new Service Graph starting from a Service Graph enriched by some Network Security Functions, already deployed and potentially configured. Consequently, in this scenario the *ADP* module can receive as input:

1. the list of medium-level Network Security Requirements which must be satisfied and which could come from the high-level user specifications;

2. the list of the selected Network Security Functions to satisfy the corresponding Network Security Requirements;

3. a Service Graph, in which some Network Security Functions are already present with a configuration previously computed or to be later configured by the framework; it could be the result of a manual operation or an antecedent run of the *ADP* element on a original Service Graph without any security property. The presence of these Network Security Functions and, if present, their configuration cannot be modified by the framework or they are subject to modifications depending of user's preferences. In addition, the service designer can provide some possible indications about places where the new Network Security Functions must or must not be placed, if the service designer has some sufficient security knowledge to take these decisions.

In this situation the output could be:

- in case of success, a new Service Graph where additional Network Security Functions have been optimally allocated in the Allocation Graph, while keeping the already existing NSFs, and their configuration is automatically created to satisfy the input medium-level Network Security Requirements. It is worth mentioning that, if the NSFs already present in the input Service Graph are provided to the framework without any configuration, it can be computed by the *ADP* module without any modification about their position in the service;

- in case of failure, a non-enforceability report to notify the service designer about the impossibility to compute the Service Graph and to underline which problems occurred. The main reasons for which in this scenario the *ADP* element could not succeed in computing the result could be that the already present Network Security Functions or their previous configuration do not allow a satisfaction of all the input security requirements, if the user does not want to modify them, or the Allocation Places are not sufficient to allocate the new needed NSFs to satisfy the requirements.

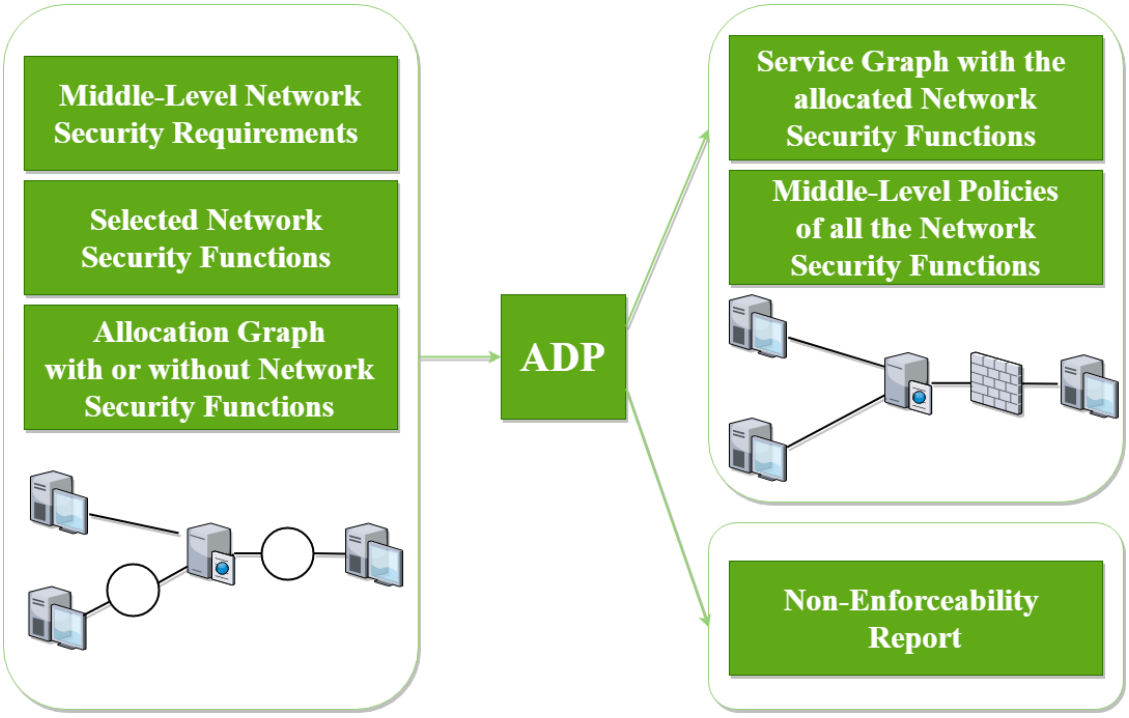


Figure 5.4. Third scenario of *AOC*

A third possible usage scenario of the *ADP* element is the scenario illustrated in Figure 5.3, where a service designer who feels confident about his knowledge in the security field wants to define all the possible placeholders where a Network Security Function can be installed by the framework. In this case, the *ADP* module can receive as input:

1. the list of medium-level Network Security Requirements which must be satisfied and which could come from the high-level user specifications;

2. the list of the selected Network Security Functions to satisfy the corresponding Network Security Requirements;

3. directly an Allocation Graph, where the service designer specifies not only the network functions which compose the service, but also all the possible Allocation Places where the Network Security Functions can be automatically allocated. Additionally, he can immediately introduce the presence of some NSFs with a manually computed configuration or to be later configured by the framework. The presence of these Network Security Functions and, if present, their configuration cannot be modified by the framework or they are subject to modifications depending of user's preferences. The framework cannot decide to put the selected NSFs in other places than the Allocation Places specifically provided by the service designer.

In this situation the output could be:

- in case of success, a Service Graph where the Network Security Functions have been optimally allocated in the Allocation Graph, while keeping the already existing NSFs, if some of them were already present, and their configuration is automatically created to satisfy the input medium-level Network Security Requirements. It is worth mentioning that, if the NSFs already present in the input Allocation Graph are provided to the framework without any configuration, it can be computed by the *ADP* module without any modification about their position in the service;

- in case of failure, a non-enforceability report to notify the service designer about the impossibility to compute the Service Graph and to underline which problems occurred. The main reasons for which in this scenario the *ADP* element could not succeed in computing the result could be that the already present Network Security Functions or their previous configuration do not allow a satisfaction of the input security requirements, if the user does not want to modify them, or the Allocation Places specifically provided by the designer are not sufficient to allocate the new needed NSFs to satisfy all the requirements.

The scenario illustrated in Figure 5.5 is, instead, a special case, where the goal is the verification of an already existing Service Graph according to a set of Network Security Requirements to satisfy, without allocating new Network Security Functions. This task is typically performed when the service designer wants to understand if a previous computation of the *ADP* element is sufficient to satisfy an extended set of security requirements, without immediately having to place and configure new Network Security Functions, which would require more time and resources; another use case is when the designer only requests VEREFOO to automatically compute the configuration of the NSFs, without changing their allocation on the logical topology.
Based on this premise, for this purpose the *ADP* module can receive as input:

1. the list of the medium-level Network Security Requirements which must be satisfied and which could come from the high-level user specifications;

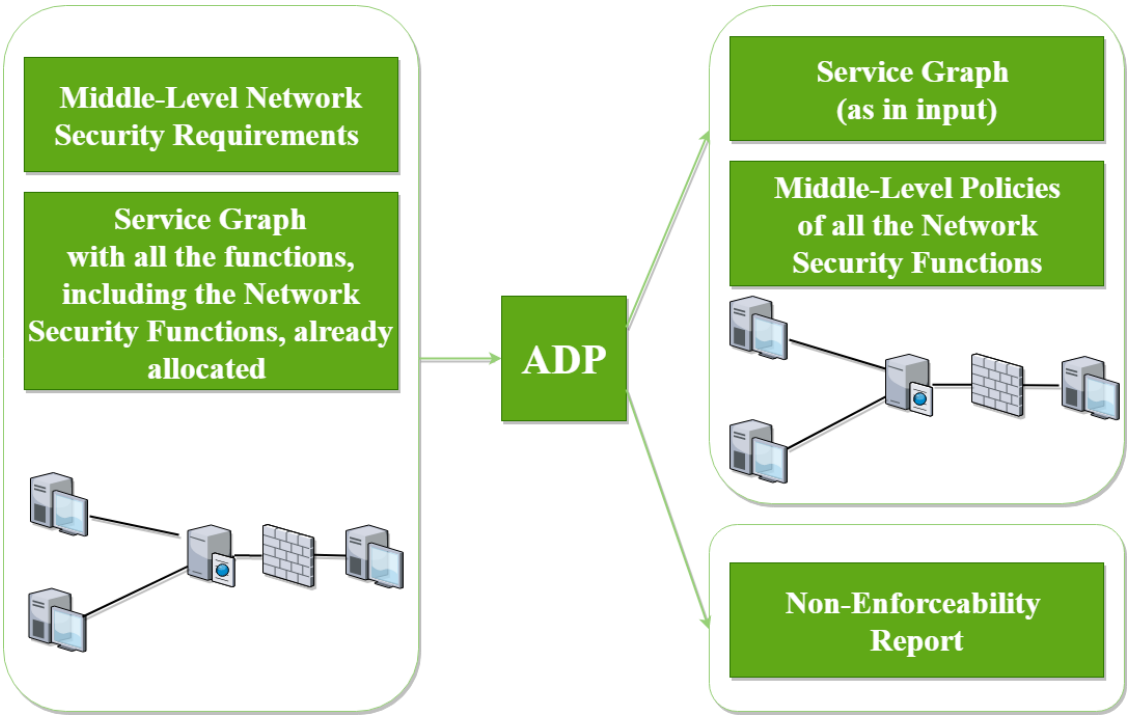


Figure 5.5. Forth scenario of *AOC*

2. a Service Graph, where all the nodes are already characterized by a specific service network function or Network Security Function, whose configuration can be either empty, if it should be filled by VEREFOO, or already defined, if computed with a previous run of the framework or manually.

In this situation the output could be:

- in case of success, a notification about the positive outcome is presented to the service designer and, if requested, the configuration of the already allocated Network Security Functions is computed with respect to the specified medium-level Network Security Requirements to satisfy;
- in case of failure, a non-enforceability report to notify the service designer about the impossibility to satisfy the Network Security Requirements with the provided Service Graph or the impossibility to compute a proper configuration of the provided Network Security Functions. The main reasons for which in this scenario the *ADP* element could not succeed in computing the result could be that the new set of Network Security Requirements is more restrictive than the original one of a previous run of the framework or the provided Network Security Functions are not sufficient to satisfy the requirements.

5.4.2 Automatic VNFs Placement

The main goal of the *Automatic VNFs Placement* task, illustrated in Figure 5.6, is to deploy the antecedently computed Service Graph on a physical network, characterized by real servers with possible limitations in terms of CPU usage, RAM

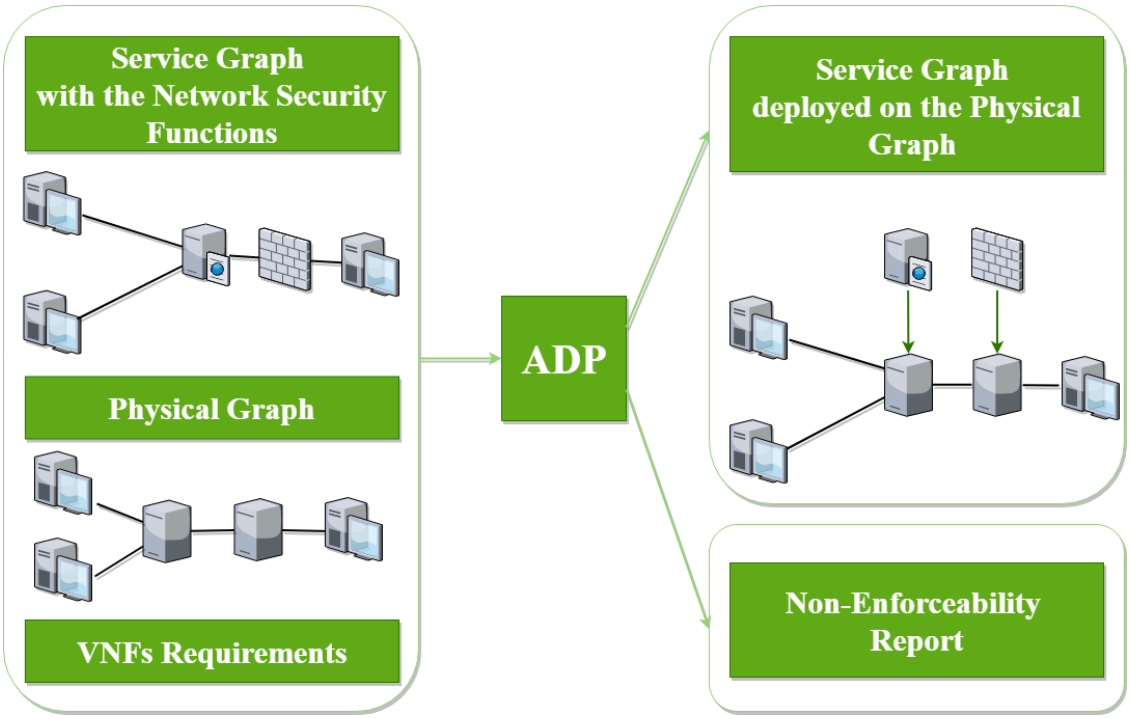


Figure 5.6. Scenario of *AVP*

availability, number of deployable network functions and performance. In the typical scenario of the *AVP* task, the *ADP* module can receive as input:

1. a Service Graph, where the nodes are already characterized by a service network function or a Network Security Function, whose configuration has been already computed manually or with a previous run of the framework;
2. a Physical Graph, which represents the characteristics (e.g. CPU power, RAM) of the physical servers of the real network to be administrated and their connections;
3. a set of VNFs requirements, representing the deployment constraints of each VNF which implements a specific service network function or Network Security Function; each VNF could, in fact, require a minimum amount of resources to be deployed on a server of the Physical Graph, where potentially other VNFs could be already installed consuming part of the total amount of computational resources. Other requirements could be, then, related to the maximum latency of the links that interconnect two servers on which two adjacent functions of the Service Graph are deployed.

In this situation the output could be:

- in case of success, the placement of all the functions of the Service Graph on the servers of the Physical Graph, so that the logical interconnections of the nodes in the Service Graph are respected and modelled as virtual links between the VNFs;

- in case of failure, a non-enforceability report to notify the service designer about the impossibility to deploy the VNFs in the Physical Graph because of lack of resources, like number of servers or total CPU and RAM availability.

This task can be combined with any of the previous scenario described in Subsection 5.4.1. This way, in just one step the service designer can get the configuration of the selected Network Security Functions based on the required medium-level Network Security Functions, perform the formal verification of all the security constraints and finally deploy the VNFs implementing the functions of the Service Graph on the physical infrastructure.

5.5 Design and development of ADP module

This thesis focused on the design and development of the *ADP* module of VEREFOO, refining an existing framework which was already able to perform the Network Security Requirements verification in Service Function Chains, a basic auto-configuration of firewalls to satisfy medium-level requirements and the deployment of VNFs on the physical infrastructure, but with not good performance in the first two features, on which the focus of this thesis has been.

The goals have been to redesign the critical first-order logic formulas of the MaxSMT problem, to implement new components like the Allocation Graph and to change a consistent part of the original framework to be compliant with the new designed formulas and the introduced data structures, so that a deeper independence between the forwarding rules (i.e. establishing the previous and next hops) and the intrinsic characteristics of the functions (i.e. the Access Control List of a packet filter) could be reached and better performance could be achieved.

The rest of the thesis presents the following work on the ADP module:

- **Chapter 6** on one side describes the design and the implementation of the Allocation Graph, on the other side the model of the new first-order logic formulas and the corresponding implementation in a z3 model of the forwarding rules in this newly logical topology;
- **Chapter 7** describes how first-order logic formulas and the implementation of the medium-level reachability and isolation Network Security Requirements were redesigned, focusing on their implementation in the Allocation Graph, the refined aspects and particularly on their extensions (i.e. verification of requirements between any pair of clients or server). It also presents the modifications made to the formulas related to wildcards to solve some existing misconfigurations and improve the performance;
- **Chapter 8** describes how the auto-configuration of the packet filter was extended, to achieve a complete distribution of the policy rules in a larger set of firewalls and with the wildcards feature rehabilitated, and how the allocation of the packet filters has been performed in the newly created Allocation Graph.

Furthermore, **Appendix B** describes the design of REST-based APIs to interface with the *ADP* module of the framework and the implementation of the corresponding RESTful web service.

Chapter 6

Allocation Graph and Forwarding Rules

In this chapter, a complete overview of the logical topologies which the framework can accept as input – the Service Graph or the Allocation Graph – is presented. First of all, a description of their implementations in the XML schema and in the framework is provided, with the assumption that for the Service Graph these components are inherited from a previous version of the framework.

Moreover, a newly designed formal model of the two graphs for the first scenario of the *Automatic Orchestration and Configuration* task, described in Section 5.3, will be presented alongside; the other scenarios represent specific cases of the first one and the models can be partially different, but they can be easily retrieved from the one here described.

About the Allocation Graph, since it is a new feature introduced in the framework, the thesis contributed in all the aspects of its implementation inside VEREFOO, contributing to define an automatic way to generate it from an input Service Graph which the service designer can define by means of an XML file.

Then, the design and implementation of the first-order logic formulas of the forwarding rules for Allocation Places are illustrated; their goal is to allow the transit of the packets between the nodes of the Allocation Graph from a source and a destination of a Network Security Requirement, taking in consideration both the scenarios in which a packet filter capability has been installed on an Allocation Place by the framework during the optimization phase or this placeholder node simply behaves as a forwarder.

6.1 Service Graph

6.1.1 Description of the Service Graph concept

Exploiting the novel paradigm of Software-Defined Networks through which it is possible to decouple the logical view of the network from the physical hosts as it was illustrated in Section 2.2, it is possible to firstly separately manage the two

planes and in a second time a service designer, who uses the framework, is allowed to consequently deploy the logical topology on the real infrastructure.

In this scenario, a Service Graph, generalization of the *Service Function Chain* concept illustrated in Section 2.1, represents the logical topology of an end-to-end service, characterized by a set of network functions – e.g. traffic monitor and web cache – that do not enforce any security defence but simply contribute to the definition of the service.

In the developed framework, a selection of network functions that the service designer can exploit in order to define a Service Graph were inherited by Verigraph. Some of these functions do not influence the configuration of the Network Security Functions given the input security requirements, while the behaviour of others has an impact which is already defined. Moreover, each network function is characterized by a number of functionalities, which contribute to the definition of the complete service by means of their composition.

A first observation is that the switch, a network function which exclusively forwards the incoming packets to an out-port selected by means of a forwarding table, is a low-level function; consequently, the service designer does not include it in the Service Graph, because the switch does not contribute to the definition of the service. On the other hand, this function can be exploited by the framework to be temporarily placed on an Allocation Place, when the optimizer establishes that a firewall is not needed in that position.

Then, the other network functions considered in the selected portfolio of Verigraph are characterized by the following classes of functionalities:

- functionalities such as *traffic monitoring* can change the value of internal counters and to notify the network administrator every time the inspection performed on a packet they have received requires an intervention. However, they do not modify the received packets and they can send them to all the possible meaningful directions; so the presence of a traffic monitor does not influence the configuration of the firewall policies.

- functionalities such as *load balancing* decide where the input traffic flow must be forwarded exploiting a specific logic with internal decisions, such as the current load of each server of the cluster. However, also a load balancer does not influence how the firewall policies are defined, because the rules deal with the end points of the network and we cannot say at priori to which server a load balancer will forward a packet. Consequently, it is as if the load balancer has to send a packet to every server of the cluster to perform the requirements verification.

- functionalities such as *NAT* are able to modify some header fields, like replacing private IP addresses with public ones. Because of this action, the rules which should be configured on a firewall could be different, according to the position in the Service Graph the packet filter is allocated; nevertheless, the input security requirements do not need any further modification, because they provide constraints about kinds of traffic flow between pair of end points.

- functionalities such as *web caching* takes forwarding decisions according to information which is typically external from the IP quintuple values, because the web cache, for example, analyses the URL of the resource which a client has requested to a web server and checks if this content is already present in itself. However, parameters such as URLs concern the application layer of the ISO/OSI stack whereas the packet filter firewalls base their forwarding behaviour on IP addresses and ports; consequently, also in the web cache case it is not possible to establish when a packet is dropped and a packet filter would not be needed.

6.1.2 Model of the Service Graph

The model of a Service Graph, received by the framework as input and where no firewall instance has still been allocated, is the following:

$$G_S = (N_S, L_S)$$

A Service Graph is, actually, characterized by two sets:

1. N_S is the set of all the network nodes of the Service Graph and it can be formalized as

   $$N_S = E_S \cup S_S$$

   In more details, E_S is a set characterized by the edge terminals of the service; with this term, not only a client or a server are referred, but also subnetworks where multiple end points are represented as a single one. Instead, S_S is a second set and it is composed by service functions (e.g. web cache, NAT), which could be needed to define a complete end-to-end service. Each element of N_S is uniquely identified by a non-negative integer number k.

2. L_S is the set of the links interconnecting a pair of elements of N_S, i.e. $l_{ij} \in L_S$, $i \neq j$ implies that $n_i \in N_S$ is directly connected to $n_j \in N_S$; the Service Graph is, in fact, modelled as a directed graph.

It is worth underlining that these model does not allow by itself the allocation of the firewalls, because every node of the graph is already characterized by a specific function. For this reason, an automatic transformation into an Allocation Graph will be needed, before the optimizer engine can establish the allocation schema and the configuration of the firewall instances.

6.1.3 Implementation of the Service Graph in the XML schema

In the XML schema, the *graph* element models a Service Graph which can be defined by the service designer and received as input by VEREFOO. It is characterized

by a unique numeric ID, a boolean attribute called *serviceGraph* and a sequence of *node* elements. The presence of the identifier allows the user to specify more than one input logical topology if he wants to deploy them simultaneously on the same physical infrastructure, as it is shown in Listing 6.1. Then, the *serviceGraph* attribute must have the value *true* if the XML file represents a Service Graph, because the *false* value – which is the default value – corresponds to an Allocation Graph.

```
<graphs>

        <graph id="0">
        ...
        </graph>

        <graph id="1">
        ...
        </graph>

</graphs>
```

Listing 6.1. XML example of multiple Service Graphs

Each *graph* element is, then, composed by a set of *node* elements; each one of them represents the basic unit of the logical topology on which either the service designer can specify a network function or the framework itself can allocate a firewall instance.

Each *node* is characterized by :

- the *name* attribute, representing the node unique name, which can be a string internally mapped to an IP address or directly a network address;
- the *functional_type* attribute, representing the function which is assigned to the specific node of the Service Graph;
- a list of *neighbour* elements, each one of which represents an adjacent node towards which a packet can be directly sent. In this case, to model a bidirectional connection, it is important to specify each node of the pair as neighbour of the other;
- a *configuration* element, which represents the behaviour of the installed network function, with all the details needed by the framework to further define the proper hard constraints of the MaxSMT problem.

A complete example of an XML file describing a Service Graph is provided by means of Listing 6.2, where the graph showed in Figure 6.1 is represented.

```
<graph id="0" serviceGraph="true">
        <node functional_type="WEBCLIENT" name="10.0.0.1">
                <neighbour name="20.0.0.1"/>
                <configuration description="WebClient_Description"
                    name="configurationWC1">
                    <webclient nameWebServer="30.0.0.1"/>
```

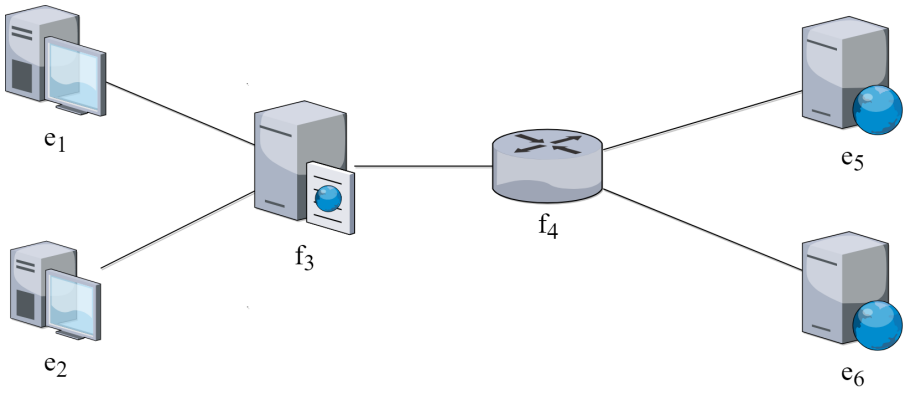


Figure 6.1. Graphical example of a Service Graph

```
        </configuration>
</node>

<node functional_type="WEBCLIENT" name="10.0.0.2">
        <neighbour name="20.0.0.1"/>
        <configuration description="WebClient_Description"
            name="configurationWC2">
                <webclient nameWebServer="30.0.0.2"/>
        </configuration>
</node>

<node functional_type="CACHE" name="20.0.0.1">
        <neighbour name="10.0.0.1"/>
        <neighbour name="10.0.0.2"/>
        <neighbour name="20.0.0.2"/>
        <configuration description="Cache_description"
            name="configurationCache">
                <cache>
                        <resource>10.0.0.1</resource>
                        <resource>10.0.0.2</resource>
                </cache>
        </configuration>
</node>

<node functional_type="NAT" name="20.0.0.2">
        <neighbour name="20.0.0.1"/>
        <neighbour name="30.0.0.1"/>
        <neighbour name="30.0.0.2"/>
        <configuration description="NAT_Description"
            name="configurationNAT">
                <nat>
                        <source>10.0.0.1</source>
                        <source>10.0.0.2</source>
                </nat>
        </configuration>
</node>

<node functional_type="WEBSERVER" name="30.0.0.1">
        <neighbour name="20.0.0.2"/>
```

```
                <configuration description="WebServer_description"
                    name="configurationWS1">
                        <webserver>
                                <name>30.0.0.1</name>
                        </webserver>
                </configuration>
        </node>

        <node functional_type="WEBSERVER" name="30.0.0.2">
                <neighbour name="20.0.0.2"/>
                <configuration description="WebServer_description"
                    name="configurationWS1">
                        <webserver>
                                <name>30.0.0.2</name>
                        </webserver>
                </configuration>
        </node>

</graph>
```

Listing 6.2. XML example of a Service Graph

6.2 Allocation Graph

6.2.1 Description of the Allocation Graph concept

The Service Graph provided by the service designer is automatically processed to create an internal representation called *Allocation Graph*, which is a core novelty this thesis work introduced. If the service designer does not specify any specific constraint, an *Allocation Place* is created replacing any link between a pair of nodes, such as end points or network functions; this element is a placeholder position where the MaxSMT solver, after solving the optimization problem, could establish if a firewall is needed to be allocated so as to achieve the optimal solution in terms of allocation schema.

Nevertheless, it is legitimate that the service designer has technical knowledge about the configuration of security defenses, if for example he also covers the role of security manager in the company of a Service Provider. In this case, he has the faculty to impose that a firewall should be placed in a specific Allocation Place, preventing the solver from removing it; secondly, the user can forbid the allocation of a firewall between a pair of nodes, if he feels that it would be useless to place an instance there. These additional features improve the capabilities of the framework, because this way it can be exploited with multiple strategies, for instance allowing to reduce the solution space by forbidding some firewalls and thus decreasing the execution time even though it could lead to an unsatisfiability of the optimization problem. Furthermore, forcing a firewall in a specific position is a useful feature to represent networks where not only virtualized functions are deployed, but even some hardware appliances are still present; in this scenario, these additional elements are perfectly represented by means of this tool.

Furthermore, if the service designer has both a clear idea how the final Service Graph would be computed and which are the only positions where the allocation of a firewall is a meaningful choice according to his technical knowledge, then he can decide to immediately introduce an Allocation Graph to the framework. An extreme case of this scenario occurs when the service designer creates a logical topology where only Allocation Places are present, as in Figure 6.2. This Allocation Graph can be, actually, exploited to understand which are the best allocation schema and configuration of the firewalls to install in the final service while ignoring all the network functions which do not have any impact on the output of the framework; it is, besides, the perfect scenario where the service designer can focus only on the Network Security Functions without caring about other middleboxes. For these reasons, this Allocation Graph is the most suitable to carry out performance tests, because it is possible to get results which exclusively depend on the the clauses of the MaxSMT problem related to firewalls and the Network Security Requirements – which nonetheless are the most critical and essential part of the framework.

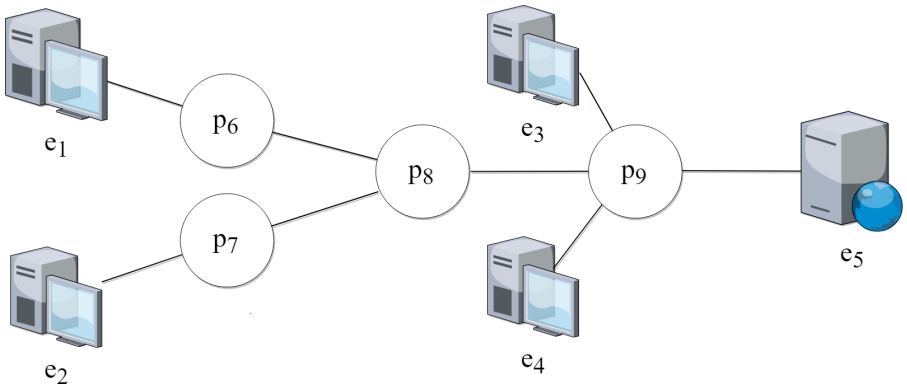


Figure 6.2. Allocation Graph with only Allocation Places between end points

Despite these benefits which are provided by a manual contribution in the configuration or creation of an Allocation Graph, nevertheless, it is important to underline that a consequence of these features could be a solution that is not optimal, since some combinations of the values assigned to the MaxSMT internal variables are pruned, even though they could be acceptable or even optimal, by means of heuristic choices taken by the user. It is, however, a possibility which the framework provides so that the user is offered a large set of options among which to choose according to his preferences and a trade-off between optimality and performance.

Each Allocation Place of this graph can have two possible roles at the beginning of the framework execution:

- if the Allocation Place is not collocated in any path between sources and destinations of the specified Network Security Requirements, then it remains empty and can be eliminated from the topology, without having any impact on the performance of the framework because in the MaxSMT problem instance no clauses are defined for it;

- if the Allocation Place is collocated in at least one path between sources and destinations of the specified Network Security Requirements, then the framework must consider the possibility to allocate a firewall in this position and a set of hard and soft constraints are introduced in the MaxSMT problem instance.

After the computation of the MaxSMT problem, for each Allocation Place two possible results could have been established:

- if a firewall has been allocated, the Allocation Place is kept in the output of the framework and is configured with a Filtering Policy which establishes, for each received packet, if it must be forwarded or dropped;
- if no firewall has been allocated, the Allocation Place has a simple forwarding behaviour, since it forwards each received packet to all the possible out-ports, and it can be removed by means of a post-processing task.

After each node of the Allocation Graph is finally assigned a specific role, the corresponding virtual implementation can be consequently deployed to the substrate servers of a Physical Graph. The separation of the two phases provides the fundamental independence between the required capabilities and the functions which represent their implementation; in fact, the choices which can be made about the logical topology of the Service or Allocation Graph are not influenced by the fact that the physical infrastructure is formed by hardware appliances or by normal servers on which Virtual Network Functions can be installed. In the first case, the assumption is that each physical middlebox can be associated with one logical node of the Allocation Graph, while in the second case it is possible to deploy more VNFs contemporary on the same server, reducing the amount of traffic on the links and the effective number of required servers.

6.2.2 Model of the Allocation Graph

Considering the scenario in which an Allocation Graph is automatically generated by a Service Graph specified as input by the service designer, the formal model of the Allocation Graph is:

$$G_A = (N_A, L_A)$$

G_A is characterized by two sets, as for the G_S model:

1. N_A is the set of all the nodes of the Allocation Graph, independently from their role and behaviour in the network and in the service, each one of which is characterized by a unique identifier.
2. L_A is the set of the links interconnecting a pair of elements belonging to N_A set, i.e. $l_{ij} \in L_A$, $i \neq j$ implies that $n_i \in N_A$ is directly connected to $n_j \in N_A$; as it is evident from this definition, also the Allocation Graph is modelled as a directed graph.

Respect to G_S, the model of the set of nodes in G_A is more complex because of the presence of the newly introduced Allocation Places. Actually, this set is modelled as:

$$N_A = E_A \cup S_A \cup P_A$$

The three sets which compose N_A are the following ones:

1. E_A is the set of the end points of the topology. Since the automatic generation of the Allocation Graph does not modify the end points, then:

$$E_A = E_S$$

2. S_A is the set of the service functions requested by the service designer. Since the automatic generation of the Allocation Graph does not modify the behaviour of these functions, then:

$$S_A = S_S$$

3. P_A is the set of the Allocation Places where a firewall can be potentially placed.

For each $p_k \in P_A$, the function *allocated*, when applied to this element, returns a boolean true result if a firewall instance has been effectively allocated on p_k, false if the Allocation Place has a simple forwarding behaviour when the optimizer engine establishes that there is no need – or it is not feasible because of other constraints – to place a firewall instance in that position. By default, the returned value is false so that, if possible, no firewall is allocated on the logical topology; nevertheless, the value can be forced to true by the service designer or by the optimizer engine when needed to satisfy the input Network Security Requirements.

Furthermore, for every $l_{ij} \in L_S$, $i \neq j$, the service designer has the faculty to perform two different actions which have an impact on the automatic generation of the Allocation Graph:

- *forbidden(l_{ij})* forbids the creation of an Allocation Place correspondent to this link, so that no firewall instance will be placed by the optimizer engine in this position. A consequence, however, could be the potential unsatisfiability of the MaxSMT problem;
- *forced(l_{ij})* obligates the allocation of a firewall in this position, despite this could lead to an unoptimized solution.

The two actions can never be performed together in the same $l_{ij} \in L_S$, because otherwise they would be in conflict.

According to the choices made by the service designer, the other constraints relative to the population of P_A, L_A sets and to the values returned by the *allocated* function when applied to a $p_k \in P_A$ are the following:

$$\begin{aligned} \forall\, l_{ij} \in L_S.\, \neg forbidden(l_{ij}) \wedge \neg forced(l_{ij}) \\ \implies p_{ij} \in P_A \wedge l_{ip_{ij}} \in L_A \wedge l_{p_{ij}j} \in L_A \end{aligned} \tag{6.1}$$

$$\forall\, l_{ij} \in L_S.\ forbidden(l_{ij}) \implies l_{ij} \in L_A \tag{6.2}$$

$$\begin{aligned} \forall\, l_{ij} \in L_S.\ forced(l_{ij}) \implies p_{ij} \in P_A \wedge l_{ip_{ij}} \in L_A \\ \wedge\, l_{p_{ij}j} \in L_A \wedge allocated(p_{ij}) = true \end{aligned} \tag{6.3}$$

The three Formulas 6.1, 6.2 and 6.3 are mutually exclusive: according to the choice made by the user, some elements are defined in the corresponding sets of the Allocation Graph. Particularly, Formula 6.1 is exploited to represent the typical scenario, where for each link $l_{ij} \in L_S, i \neq j$ an Allocation Place is created because no additional constraints are defined by the user. In this scenario, only the Allocation Place is created, while the allocation of the firewall could be decided upon different constraints – typically some soft clauses –.

It is possible to underline that among these three possibilities only Formula 6.3 sets a definitive value for the *allocated* function when applied to a specific Allocation Place; in this case, solving the corresponding MaxSMT problem requires to satisfy this hard constraint.

6.2.3 Implementation of the Allocation Graph in the XML schema

The implementation of the Allocation Graph in the XML schema was created considering the Service Graph structure as a starting point. With respect to the Service Graph, actually, the main difference is the possibility to specify *node* elements without either any *functional_type* attribute or any *configuration* internal element, because these values will be established by the framework after solving the MaxSMT problem. Listing 6.3 shows a simple example of an Allocation Place, which can be added to an XML file by the service designer or automatically by the framework itself.

```
<node name="10.0.0.1">
        <neighbour name="10.0.0.2" />
        <neighbour name="10.0.0.3" />
</node>
```

Listing 6.3. XML example of an Allocation Place

Inside the *Constraints* element of an input XML file, the service designer can specify the *forced* and *forbidden* action on a pair of nodes by adding an *AllocationConstraints* element, which represents another novelty introduced during the thesis work. It contains a set of *AllocationConstraint* elements, each one of which is characterized by:

- a *type* attribute, which can have the two alternative string values *forced* or *forbidden*;
- a *nodeA* attribute, which identifies the first extremity node of the link on which the constraint is defined;

- a *nodeB* attribute, which identifies the second extremity node of the link on which the constraint is defined.

To provide an example of how the automatic generation of an XML file describing an Allocation Graph is performed, it is possible to consider the XML file of the Service Graph presented in Listing 6.2. The supposition is that, alongside with this Service Graph, the service designer provides the set of constraints, related to the allocation of firewall instances, described in Listing 6.4.

```
<Constraints>
        <NodeConstraints/>
        <LinkConstraints/>
        <AllocationConstraints>
                <AllocationConstraint type="forbidden" nodeA="10.0.0.2"
                    nodeB="20.0.0.1"/>
                <AllocationConstraint type="forced" nodeA="20.0.0.2"
                    nodeB="30.0.0.1"/>
        </AllocationConstraints>
</Constraints>
```

Listing 6.4. XML example of a set of Allocation Constraints

The XML file automatically generated by the framework, describing the Allocation Graph which is then modelled in the MaxSMT problem, is provided in Listing 6.5 and graphically represented in Figure 6.3. It is worth noticing how also the elements which were present in the original XML file have been modified, such as the *neighbour* elements; moreover, the firewall which has been introduced is still without configuration, which will be established only after solving the optimization problem in a second moment.

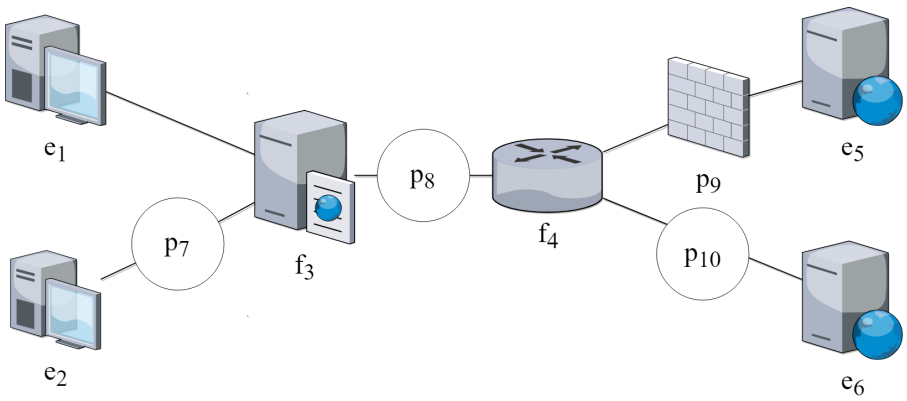


Figure 6.3. Graphical example of the automatically generated Allocation Graph

```
<graph id="0">
        <node functional_type="WEBCLIENT" name="10.0.0.1">
                <neighbour name="20.0.0.1"/>
                <configuration description="WebClient_Description"
                    name="configurationWC1">
                        <webclient nameWebServer="30.0.0.1"/>
                </configuration>
        </node>
```

```
<node functional_type="WEBCLIENT" name="10.0.0.2">
       <neighbour name="40.0.0.1"/>
       <configuration description="WebClient_Description"
           name="configurationWC2">
               <webclient nameWebServer="30.0.0.2"/>
       </configuration>
</node>

<node name="40.0.0.1">
       <neighbour name="10.0.0.2"/>
       <neighbour name="20.0.0.1"/>
</node>

<node functional_type="CACHE" name="20.0.0.1">
       <neighbour name="10.0.0.1"/>
       <neighbour name="40.0.0.1"/>
       <neighbour name="40.0.0.2"/>
       <configuration description="Cache_description"
           name="configurationCache">
               <cache>
                       <resource>10.0.0.1</resource>
                       <resource>10.0.0.2</resource>
               </cache>
       </configuration>
</node>

<node name="40.0.0.2">
       <neighbour name="20.0.0.1"/>
       <neighbour name="20.0.0.2"/>
</node>

<node functional_type="NAT" name="20.0.0.2">
       <neighbour name="40.0.0.2"/>
       <neighbour name="40.0.0.3"/>
       <neighbour name="40.0.0.4"/>
       <configuration description="NAT_Description"
           name="configurationNAT">
               <nat>
                       <source>10.0.0.1</source>
                       <source>10.0.0.2</source>
               </nat>
       </configuration>
</node>

<node functional_type="FIREWALL" name="40.0.0.3">
       <neighbour name="20.0.0.2"/>
       <neighbour name="30.0.0.1"/>
       <configuration description="Firewall_Description"
           name="configurationFW">
               <firewall />
       </configuration>
</node>

<node name="40.0.0.4">
       <neighbour name="20.0.0.2"/>
       <neighbour name="30.0.0.2"/>
```

```
        </node>


        <node functional_type="WEBSERVER" name="30.0.0.1">
               <neighbour name="40.0.0.3"/>
               <configuration description="WebServer_description"
                   name="configurationWS1">
                      <webserver>
                             <name>30.0.0.1</name>
                      </webserver>
               </configuration>
        </node>

        <node functional_type="WEBSERVER" name="30.0.0.2">
               <neighbour name="40.0.0.4"/>
               <configuration description="WebServer_description"
                   name="configurationWS1">
                      <webserver>
                             <name>30.0.0.2</name>
                      </webserver>
               </configuration>
        </node>

</graph>
```

Listing 6.5. XML example of an automatically generated Allocation Graph

6.2.4 Implementation of the Allocation Graph in the framework

In the framework, the Allocation Graph can be automatically generated by the newly developed *AllocationGraphGenerator* class, given an input Service Graph and a set of allocation constraints; if, instead, the service designer immediately specifies an Allocation Graph as input, the role of this class is not needed.

Each node of the Allocation Graph is internally represented as an object of the *Allocation Node* class. The *Allocation Node* class is an extension of the JAXB-annotated *Node* class corresponding to the homonymous element in the XML schema; the purposes of its creation are to internally implement the *hashCode* and *equals* methods so that the instances can be used as keys in maps or in sets and to store all the necessary information for the creation of forwarding rules and routing tables.

In fact, in order to achieve independence by the formulas for the forwarding rules which will be described in Section 6.3 and the data on which they are applied, four maps are provided inside each instance of this class:

- a map called *leftHops*, where the key is the node from which the packet has been received and the value is the set of the nodes to which the packet can be sent to reach the destinations of the isolation and reachability requirements;
- a map called *rightHops*, where the key is the node to which the packet has been sent and the value is the set of the nodes from which the packet can

have been received traversing the network to reach the destinations of the isolation and reachability requirements;

- a map called *lastHops*, where the key is the source of a security requirement and the value is the set of the nodes which can be the last hops to reach the destination;
- a map called *firstHops*, where the key is the destination of a security requirement and the value is the set of the nodes which can be the first hops to reach it starting from a corresponding source.

According to the specific network capability which could be deployed on a node of the Allocation Graph, the forwarding rules will be different, but they will use the same data, stored in these maps thanks to a recursive visit of the topology done before the installation of the functions.

In addition, the *NFAllocationManager* class can manage all the information about the Allocation Nodes and it implements both the algorithmic decision thanks to which a function is deployed on a node and the invocation of the specific methods which can create the z3 formulas after the computation of the previously explained maps.

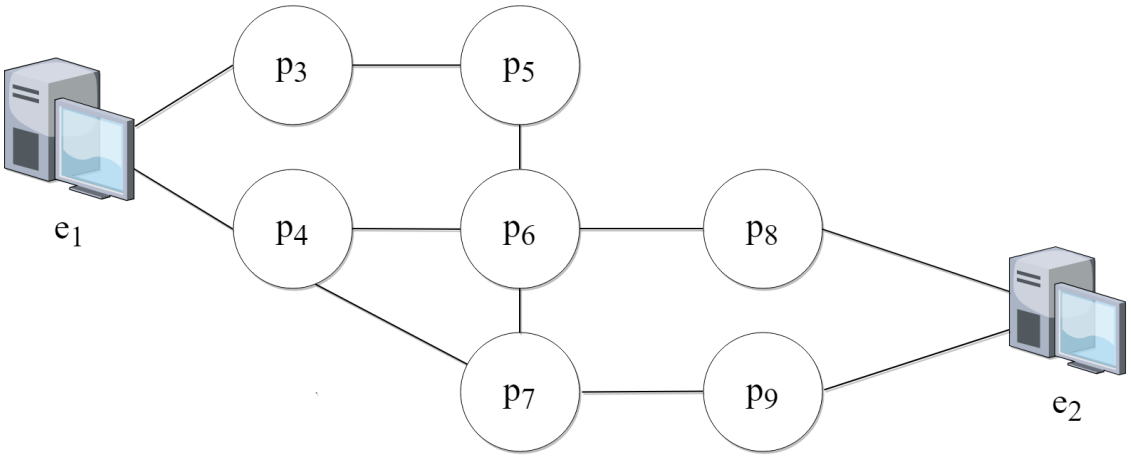


Figure 6.4. Example of an Allocation Graph to illustrate the maps of an Allocation Place

To provide a better comprehension of this feature, an example is provided by means of Figure 6.4, where the supposed Network Security Requirement to satisfy is a reachability (or indifferently isolation) requirement from the end point *e1* to the end point *e2*.

Through a recursive visit of the logical topology, the following chains of nodes are extracted as possible loop-free paths between the two hosts:

- $e_1 - p_3 - p_5 - p_6 - p_8 - e_2$
- $e_1 - p_3 - p_5 - p_6 - p_7 - p_9 - e_2$
- $e_1 - p_3 - p_5 - p_6 - p_4 - p_7 - p_9 - e_2$

- $e_1 - p_4 - p_6 - p_8 - e_2$
- $e_1 - p_4 - p_6 - p_7 - p_9 - e_2$
- $e_1 - p_4 - p_7 - p_6 - p_8 - e_2$
- $e_1 - p_4 - p_7 - p_9 - e_2$

Focusing on p_6, his *leftHops* can be p_4, p_5 and p_7 because a packet can come from them according to the effective path which is traversed, while the *rightHops* are p_4, p_7 and p_8 because they are the possible next hops of its routing table; it is possible to notice, for example, that p_5 is not a *rightHop* for p_6 because otherwise a loop in the forwarding would be created.

After the visit of the graph, the concise representation of the contents of the *leftHops* and *rightHops* maps of p_6 is provided in Listing 6.6.

```
leftHops(p6)={ p4 = [ p7, p8 ], p5 = [ p4, p7, p8 ], p7 = [ p8 ]}
rightHops(p6)={ p4 = [ p5 ], p7 = [ p4, p5 ], p8 = [ p4, p5, p7 ]}
```

Listing 6.6. Concise representation of the maps of an Allocation Node

Another useful piece of terminology which can be exploited to simplify the representation and the understanding of the contents of these maps is that, given *lH* a specific *leftHop* of p_6, the function $rightHop(lH, p_6)$ returns the set of all the possible next hops where the packets can be forwarded, after being received by p_6 from the direction of *lH*. Vice versa, the same terminology can be applied, given *rH* a specific *rightHop* of p_6, to the function $leftHop(rH, p_6)$; in this case, the returned set is composed by the nodes from which the packet can have been received, if it has been sent to *rH*.

An extended representation of the maps of the example in exam is, furthermore, provided in Listing 6.7, where the two aforementioned functions are exploited.

```
leftHops(p6) = { p4, p5, p7 }
rightHops(p4, p6) = { p7, p8 }
rightHops(p5, p6) = { p4, p7, p8 }
rightHops(p7, p6) = { p8 }

rightHops(p6) = { p4, p7, p8 }
leftHops(p4, p6) = { p5 }
leftHops(p7, p6) = { p4, p5 }
leftHops(p8, p6) = { p4, p5, p7 }
```

Listing 6.7. Extended representation of the maps of an Allocation Node

Moreover, a complete explanation of the reasons for which, given a *rightHop* or a *leftHop*, a correspondent list of nodes is associated is here provided:

- if p_6 receives a packet from p_4 as *leftHop*, then it can send this packet only to p_7 or p_8, because they are the only *rightHops* through which the destination can be reached if the packet comes from p_4;

- if p_6 receives a packet from p_5 as *leftHop*, then it can send this packet to either p_4, p_7 or p_8, because they are the only *rightHops* through which the destination can be reached if the packet comes from p_5;
- if p_6 receives a packet from p_7 as *leftHop*, then it can send this packet only to p_8, because it is the only *rightHop* through which the destination can be reached if the packet comes from p_7;
- if p_6 sends a packet to p_4 as *rightHop*, then this packet could have been received only from p_5 because it is the only *leftHop* passing from which the destination can be reached continuing through p_4;
- if p_6 sends a packet to p_7 as *rightHop*, then this packet could have been received either from p_4 or from p_5, because both of them are *leftHops* candidates passing from which the destination can be reached continuing through p_7;
- if p_6 sends a packet to p_8 as *rightHop*, then this packet could have been received from p_4, p_5 or p_7, because all of them are *leftHops* candidates passing from which the destination can be reached continuing through p_8.

Moving then the focus on the end point e_1, since both p_8 and p_9 can potentially be the last nodes traversed by a packet sent to the network by e_1 to reach the destination e_2, the content of its *lastHops* map is represented in Listing 6.8.

```
lastHops(e1) = { e2 = [ p8, p9 ] }
```

Listing 6.8. Representation of the content of the *lastHops* map

Symmetrically, since both p_3 and p_4 can potentially be the first nodes traversed by a packet sent to the network by e_1 to reach the destination e_2, the content of the *firstHops* maps of e_2 is represented in Listing 6.9.

```
firstHops(e2) = { e1 = [ p3, p4 ] }
```

Listing 6.9. Representation of the content of the *firstHops* map

6.3 Forwarding Rules

6.3.1 Design of the Forwarding Rules

In designing the forwarding rules by means of which a node decides if a received packet must be forwarded to some next hops or dropped, it is possible to highlight two different cases:

- for all the network functions which the service designer can exploit to create a Service Graph, the design of the construction of the routing tables defined in Verigraph has been kept. However, since the original formulas for forwarding rules were able to consider only loop-free paths, where at maximum one left hop and one right hop exist for a node traversed by a packet, some modifications were needed to cover a more complex scenario, where potentially each

node, including clients and servers, can be linked to a multiplicity of other nodes to every other direction;

- for all the Allocation Places, new first-order logic formulas have been introduced during this thesis work. Furthermore, since in each Allocation Place a firewall can be installed, this forwarding behaviour is shared also by the packet filtering capability.

In the following part of this subsection, the focus will be on presenting the new formulas introduced for the Allocation Places, which have been a critical phase of the modelling work of this thesis. For a better comprehension of the formulas which will be afterwards presented, useful terminology is provided in the following:

pk_0 it is the variable representing a packet;

$recv(n_1, n_2, pk_0)$ it is an expression which means that node n_2 has received the packet pk_0 from node n_1;

$send(n_1, n_2, pk_0)$ it is an expression which means that node n_1 has sent a packet pk_0 to node n_2;

$behaviour(p_k, pk_0)$ it is a variable representing the matching of the Filtering Policy of the allocated firewall with the packet pk_0 and is present only when the packet filtering capability is used on the specific Allocation Place.

The forwarding rules are modelled by the following FOL formulas, which a consider a generic packet (called *pk0* in the first-order logic) that an intermediate Allocation Place $p_k \in P_A$ should manage:

$$\begin{aligned} &\forall\, lH \in leftHops(p_k).\ (\exists\, pk_0.\ recv(lH, p_k, pk_0) \wedge behaviour(p_k, pk_0) \\ &\wedge allocated(p_k) \implies \forall rH \in rightHops(lH, p_k).\ send(p_k, rH, pk_0)) \end{aligned} \tag{6.4}$$

$$\begin{aligned} &\forall\, rH \in rightHops(p_k).\ (\exists\, pk_0.\ send(p_k, rH, pk_0) \wedge behaviour(p_k, pk_0) \\ &\wedge allocated(p_k) \implies \exists lH \in leftHops(rH, p_k).\ recv(lH, p_k, pk_0)) \end{aligned} \tag{6.5}$$

$$\begin{aligned} &\forall\, lH \in leftHops(p_k).\ (\exists\, pk_0.\ recv(lH, p_k, pk_0) \wedge\ \neg allocated(p_k) \\ &\implies \forall rH \in rightHops(lH, p_k).\ send(p_k, rH, pk_0)) \end{aligned} \tag{6.6}$$

$$\begin{aligned} &\forall\, rH \in rightHops(p_k).\ (\exists\, pk_0.\ send(p_k, rH, pk_0) \wedge\ \neg allocated(p_k) \\ &\implies \exists lH \in leftHops(rH, p_k).\ recv(lH, p_k, pk_0)) \end{aligned} \tag{6.7}$$

Formula 6.4 states that, if the Allocation Place has received a packet from a *leftHop*, a firewall is effectively allocated and its behaviour does not drop the packet, then it must be sent to every *rightHop* which allow to reach the destination, given the *leftHop* of provenance; this way all the possible paths to reach a destination are tried.

Formula 6.5 states that, if the Allocation Place has sent a packet to a *rightHop*, a firewall is effectively allocated and its behaviour does not drop the packet, then it must have been received from at least one *leftHop* from which the source could have sent a packet; it is in fact acceptable that some *leftHops* do not let packets transit because of their functionalities.

Formula 6.6 states that, if the Allocation Place has received a packet from a *leftHop* and no firewall is actually allocated, then the packet must be sent to every *rightHop* which allow to reach the destination, given the *leftHop* of provenance; in fact, the behaviour is that of a forwarder, so that all the possible paths to reach a destination are tried every time a packet is received.

Formula 6.7 states that, if the Allocation Place has sent a packet to a *rightHop* and no firewall is actually allocated, then the packet must have been received from at least one *leftHop* from which the source could have sent a packet; it is in fact acceptable that some *leftHops* do not let packets transit because of their functionalities.

Formulas 6.4 and 6.5 work in couple, similarly as Formulas 6.6 and 6.7, in order to guarantee the correctness of both the formulations of the isolation and reachability Network Security Requirements, which will be presented in Sections 7.5 and 7.6.

Actually, on one side, without Formulas 6.4 and 6.6, the optimizer engine, while trying to satisfy an isolation requirement, could make false the result of the *send* functions even when the corresponding *recv* function is true, if the only exploited forwarding rules were Formulas 6.5 and 6.7.

On the other side, without Formulas 6.5 and 6.7, the optimizer engine, while trying to satisfy a reachability requirement, could make true the result of the *send* functions even when the corresponding *recv* function is false, if the only exploited forwarding rules were Formulas 6.4 and 6.6.

6.3.2 Implementation of the Forwarding Rules

Two different ways to implement the forwarding rules through which nodes can propagate the packets they have received have been evaluated in comparison:

- they can be built using quantifiers;
- they can be built strictly using the information about left and right hops in the paths related to every requirement.

The first solution was adopted by the original version of the framework, which made an abundant use of *quantifiers*: they are a particular type of variable which allows to express a larger set of elements than just a single one inside a formula. To provide an example with the goal to clarify the meaning of a quantifier variable, we can consider the z3 assertion presented in Listing 6.10 and related to a node called *node1* with neighbours *node2* and *node3* and on which the decision to deploy a packet filter should be taken by the framework:

```
(assert (forall ((pk0 packet) (n0 node))
       (=> (and (recv n0 node1 pk0) (not
          packet_filter_used_node0))
              (and (send node1 node2 pk0) (send node1 node3
                 pk0)))))
```

Listing 6.10. z3 assertion for a forwarding example with quantifiers

The explanation of this assertion is that for each packet *pk0* which the node *node1* has potentially received from each node *n0*, if the packet filter is not installed, then it must send the same packet to another node *node2*.

This way there is no distinction about *leftHops* and *rightHops*, because the quantifier variable *n0* can represent every other instance of that type, i.e. it represents all the network nodes in the graph; the formula is, consequently, more concise and does not require the collection of the information in the maps described in Subsection 6.2.4.

However, the use of quantifiers leads to very bad performance, because in each formula involving them, all the possible variables of the same type are considered by the optimizer. Consequently, the number of solutions that z3 investigates is too huge to make large network instances feasible to be managed by the framework.

The second solution, adopted and developed by this thesis work to overcome the limitation of the original approach, requires that in the formulas of the forwarding rules each variable representing a node is not defined as a quantifier, but as the specific corresponding name of that precise node inside the z3 model which coincides with the IP address or the alternative unique name for simplicity of implementation. This is possible only thanks to the Allocation Graph implementation and the construction of the internal maps of each Allocation Node from which information about previous and next hops can be easily and efficiently retrieved when the forwarding rules must be built in a second time.

An example is provided by means of Figure 6.5 to make clearer the differences between the two solutions.

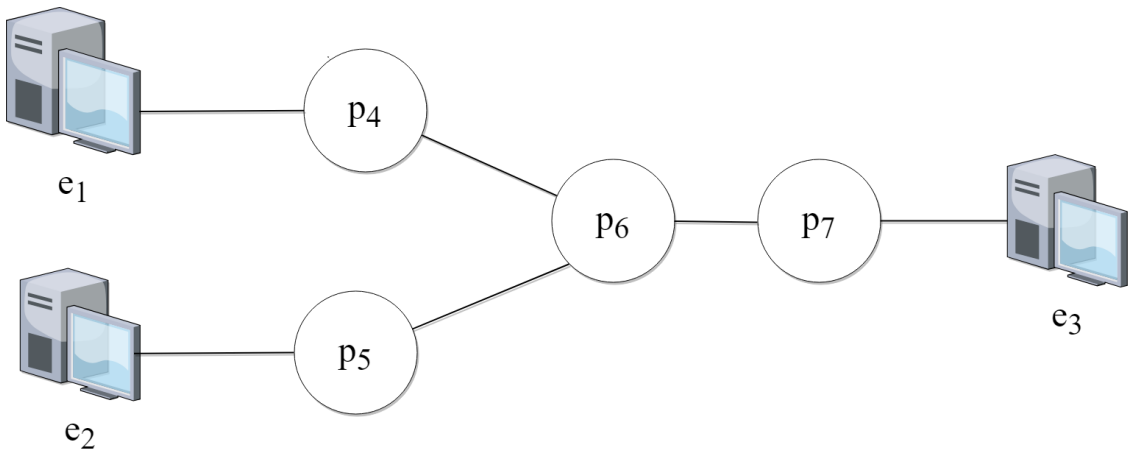


Figure 6.5. Case of study for forwarding rules

Considering exclusively the forwarding rule represented by Formula 6.4 through which the Allocation Place p_6 must decide according to the behaviour of the installed firewall functionality if a packet coming from e_1 should be sent towards e_3, it can be implemented in the framework in two different ways.

If quantifiers are used, the rule is modelled by the following formula:

$$\exists\, n_0.\ recv(n_0, p_6, pk_0)\ \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow\ send(p_6, p_7, pk_0) \quad (6.8)$$

In this formula, *n0* is a quantifier representing all the possible variables of type *node* inside the same z3 model. As a consequence, when the optimizer parses the input hard and soft constraints, Formula 6.8 is expanded creating a different formula for each one of the different node present in the network:

$$\begin{aligned}
&recv(e_1, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)\\
&recv(e_2, p_6, pk0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)\\
&recv(e_3, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)\\
&recv(p_4, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)\\
&recv(p_5, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)\\
&recv(p_6, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)\\
&recv(p_7, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_7, pk_0)
\end{aligned}$$

In this first solution, all the possible events of receiving a packet from any node of the network, including p_6 itself, are considered. The advantage of this approach is that Formula 6.8 is much compact and can be built using only information about the next hops towards a destination; actually, the quantifier is a powerful and expressive tool, which provides expressiveness and clearness. On the other hand, the performance results take a severe hit by this feature, because the optimizer is forced to consider a huge number of constraints; if the network is made by 100 nodes, than the single formula is expanded by z3 in 100 separated constraints, dramatically increasing the size of the solution space.

If the maps described in Subsection 6.2.4 are exploited instead of the quantifiers, the rule is immediately modelled by the following formulas, without needing a translation by the optimizer:

$$\begin{aligned}
&recv(p_4, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_4, pk_0)\\
&recv(p_5, p_6, pk_0) \wedge behaviour(p_6, pk_0) \wedge allocated(p_6) \Rightarrow send(p_6, p_4, pk_0)
\end{aligned}$$

This second approach is feasible because a recursive visit of the graph can provide the information that a packet, related to the requirement whose source is e_1 and destination is e_3, can be received by p_6 exclusively from p_4 or p_5 and it makes sense to send it only to p_7 to reach its real destination, avoiding to consider both useless paths which do not allow to reach possible destinations and left hops from which it is evident that a packet would never be received. More computation time to build these data structures is required, but it is always negligible in comparison to the working time of the optimizer.

After presenting this practical example, the Java implementation of Formula 6.4 is provided in Listing 6.11 to have a complete overview on how forwarding rules are

actually built in the Java APIs which exploit the z3 language to model the relative hard constraints inside the MaxSMT problem.

```
for(Map.Entry<AllocationNode, Set<AllocationNode>> entry :
   source.getLeftHops().entrySet()) {

        AllocationNode an = entry.getKey();
        Expr e = an.getZ3Name();

        BoolExpr recv= (BoolExpr) nctx.recv.apply(e, fw, p_0);
        List<Expr> list = entry.getValue().stream().map(n ->
           n.getZ3Name()).collect(Collectors.toList());
        List<Expr> sendNeighbours = list.stream().map(n ->
           (BoolExpr) nctx.send.apply(fw, n,
           p_0)).distinct().collect(Collectors.toList());
        BoolExpr[] tmp = new BoolExpr[list.size()];
        BoolExpr enumerateSend =
           ctx.mkAnd(sendNeighbours.toArray(tmp));

        if(autoplace) {
               constraints.add(ctx.mkForall(new Expr[] { p_0 },
               ctx.mkImplies(ctx.mkAnd((BoolExpr) recv,
               behaviour, used), ctx.mkAnd(enumerateSend)),
               1, null, null, null, null));
        }

}
```

Listing 6.11. Java implementation of Formula 6.4

Chapter 7

Network Security Requirements

After defining a Service Graph, the service designer can specify a set of Network Security Requirements which must be satisfied by an automatic allocation of firewalls on the generated Allocation Graph. Among all the kinds of security constraints which can be specified, this thesis focused on *connectivity requirements*, i.e. reachability and isolation properties. The isolation and reachability constraints can specify the respective need of blocking or allowing a communication between two end points or two subnetworks including a multiplicity of nodes. In the original version of the framework, however, these features were exclusively applied to an interaction established from a client to a server in a limited scenario.

For this reason, first of all a more complete model has been introduced in relation to the Network Security Requirements which deal with the packet filters rules, i.e. these security requirements are based on the components of the IP quintuple. A general description is provided in Section 7.1, while the formal model is presented in Section 7.2 and the relative implementation, inherited by the previous framework version, in the XML schema is described in Section 7.3.

An overview about how this thesis contributed to the modification of the *wildcards* feature for IP addresses is provided in Section 7.4; it is a traversal concept which is exploited not only in the components of the Network Security Requirements, but also in other aspects of the framework such as the definition of the end points of a Service or Allocation Graph and the auto-configuration of the firewall policy rules. Improving the performance which can be achieved through this feature has been a critical step of the overall work.

A second step was to remodel the first-order logic formulas and, accordingly, their implementation in the framework in order to allow interactions between end points in some scenarios where loops are present and in the scenarios where an end point is linked to multiple nodes; when a loop of links is present, in fact, if all the loop-less paths from the source to the destination are not tried it is impossible to understand if the requirement is really satisfied. The new formulas are presented in Section 7.5 for the isolation property, Section 7.6 for the reachability property.

A third step was to improve the performance by removing the quantifiers, whose problems have already been explained in Section 7.4, together with the new implementations developed to overcome these limitations.

A further extension, presented in Section 7.7, has allowed to express the Network Security Requirements for every pair of end points in the Allocation Graph or in the Service Graph, independently of their specific role (i.e. client or server), in order to model a more realistic network environment; this has been achieved by exploiting the introduction itself of the Allocation Graph, as it has been explained in Section 6.2.

Finally, as described in Section 7.8, the FOL formulas introduced to model the Network Security Requirements have allowed, as a consequence, the possibility that the same source can generate different kinds of traffic flow, characterized by different port numbers and transport-level protocols, avoiding to exploit pre-processing and post-processing tasks to manage multiple security requirements between the same pair of end points.

7.1 Description of the Network Security Requirements

The Network Security Requirements represent the second input of VEREFOO and they can be formulated by the user of the framework exploiting two different representations, according to his experience in terms of security:

1. if the service designer does not know all the security details of the configurations of the Network Security Functions to be deployed, when he has to define the input requirements, he can exploit a high-level representation, that is flexible to manage and simple to understand. In this user-friendly approach, each end point of the service is characterized by a unique identifier, that is a label which can then be mapped to an IP address. To cite an example, it is legitimate to request that the MongoDB database must not be reachable by a Tomcat server, even though their current IP addresses (and even ports) are not known, because these parameters can be established by the framework automatically, exploiting the identifiers of each end point.

2. if instead the service designer knows how security functions are actually configured and he is confident in his capabilities, he can decide to immediately exploit a medium-level representation, including details such as IP addresses and ports so that the tool can immediately use them to define the MaxSMT problem, without needing an additional operation of translation. Moreover, these representations are formulated so that they do not regard the set-up of the virtual functions, which is dependent on different vendors and can be easily computed afterwards.

In VEREFOO, the *H2M* module, described in Section 5.2, performs a translation from the high-level requirements to the medium-level representations; consequently, the *ADP* module receives exclusively the constraints expressed by means of a medium-level language and which do not need any further modification. The thesis work considered this aspect as an assumption in defining a new model for representing the security constraints.

In more details, the Network Security Requirements which this thesis analysed are the *connectivity requirements* between a pair of end points; in particular, they can be classified in two different types:

- *reachability property*, if an end point must be able to reach another one by exploiting at least an allowed path in the graph;
- *isolation property*, if an end point must not reach another one through any possible route.

In addition, the service designer can specify a *general behaviour* as input to VEREFOO. The general behaviour describes how the framework should manage all the kinds of traffic for which the service designer has not formulated any specific security requirement and it can assume three possible values:

- *whitelisting*, if all the communications for which no specific requirement is formulated should be blocked;
- *blacklisting*, if all the communications for which no specific requirement is formulated should be allowed;
- *specific*, if the way how all the communications for which no specific requirement is formulated are handled is not important for the service designer.

The third alternative, i.e. the *specific* general behaviour, represents a *Don't Care* scenario where the user of the framework is interested in enforcing a set of specific security requirements between selected pairs of end points, without caring about all the other communications. This decision on one side requires that the service designer is able to clearly identify all the kinds of traffic for which some security constraints must be satisfied in the provided Service Graph, on the other side it allows the framework to exploit this approach by reaching the goals expressed through the soft clauses of the MaxSMT problem exclusively focusing on the specific Network Security Requirements.

If a *specific* general behaviour is adopted, nevertheless, it is fundamental that the *PAN* module of VEREFOO, described in Section 5.2, performs a conflict analysis of the specific Network Security Requirements set, since the constraints specified must be conflict-free; this problem, on the other hand, is not present with the *whitelisting* or *blacklisting* approaches, because the user is allowed to respectively specify only reachability or isolation properties.

During this thesis, where the development of the *ADP* module represented the core of all the work, the assumption was that the general behaviour adopted is always *specific*, because the other two cases can be brought back to this approach by means of pre-processing tasks. For this reason, a second assumption is that the input set of Network Security Requirements is always expressed in a medium-level language and that it is conflict-free.

7.2 Model of the Network Security Requirements

The set of the specific Network Security Requirements which the service designer can personally define and must be satisfied on an input Service Graph is represented by the letter R in the formal model.

Each requirement in this set, i.e. each security requirement, is formulated exploiting the following components of a medium-level representation:

[ruleType, IPSrc, IPDst, portSrc, portDst, tranportProto]

Each expression is characterized by the following six elements:

- *ruleType* can have the values *reachability property* or *isolation property* and specifies which kind of constraints should be satisfied for a specific kind of communication;
- *IPSrc* is the source IP address of the communication for which the requirement is specified;
- *IPDst* is the destination IP address of the communication for which the requirement is specified;
- *portSrc* is the transport-level source port of the communication for which the requirement is specified;
- *portDst* is the transport-level destination port of the communication for which the requirement is specified;
- *transportProto* is the transport-level protocol of the communication for which the requirement is specified.

For the representation of the IP addresses, i.e. *IPSrc* and *IPDst*, the traditional dot-decimal notation has been exploited:

$$ip_1.ip_2.ip_3.ip_4$$

where ip_i, $\forall i \in \{1,2,3,4\}$, can be an integer in the interval from 0 to 255, extremes included, or alternatively a *wildcard* element, identified with $*$. This symbol allows to have a unique statement for both a network address and the corresponding netmask, instead of two separate elements: for example, the 20.6.8.$*$ representation can be used to express the end points that are present in the network 20.6.8.0/24, while the 30.2. $*$.$*$ representation characterizes the network 30.2.0.0/16. Further information about the new implementation of this feature, which this thesis work contributed to, is provided in Section 7.4.

The IP addresses which are specified as source and destination in the Network Security Requirements do not necessarily coincide with the IP addresses of the end points of the Service Graph; for instance, if the service designer provides a reachability property between the source 10.0.0.$*$ and the destination 30.0.0.1, then

if in the Service Graph the two nodes 10.0.0.1 and 10.0.0.2 are present two kinds of traffic should be actually allowed. For this reason, a pre-processing task already existent in the framework was exploited so that, when the formulas of reachability and isolation properties are built in the model of the MaxSMT problem, each property refers to IP addresses effectively assigned to nodes in the logical topology. In the aforementioned examples, the pre-processing task, from those input security requirements, would easily create two separate requirements, where the first has 10.0.0.1 as source, while the second has 10.0.0.2 as source.

Then, the source and destination transport-level ports *portSrc* and *portDst* can be formulated by means of a single number or an interval of numbers, considering a range from 0 to 65535. To clarify this concept, if it is required that the source 10.5.8.4 must not be able to reach the destination 20.3.6.1 is the port numbers are included in the interval [1000, 2000], the traffic flow between the two end points must be blocked if the packets are characterized by a source port number included in this interval, so that the isolation requirement is satisfied.

Finally, the *transportProto* element represents the layer-4 protocol which is used above the IP layer; the formulation can have *TCP* and *UDP* as possible values or, also for this component, the wildcard $*$. In this case, $*$ requires that the satisfiability of the property must be guaranteed considering the possibility that the source can send both TCP and UDP packets.

7.3 Implementation in the XML schema of the Network Security Requirements

The implementation in the XML schema, inherited by the previous version of the framework, of the reachability and isolation requirements is made by *Property* elements, each one of which is characterized by the following attributes:

name it is the kind of requirement (i.e. reachability or isolation);

src it is the identifier (e.g. IP address) of the source node of the requirement;

dst it is the identifier (e.g. IP address) of the destination node of the requirement;

src_port it is the number or interval of numbers for the source port;

dst_port it is the number or interval of numbers for the destination port;

l4_proto it is the type of layer 4 protocol (i.e. TCP or UDP).

A couple of examples, taken from an XML input file of the framework, are presented in the following.

Listing 7.1 states a reachability requirement according to which every node belonging to the subnetwork 10.0.0.0/24 must be able to reach the destination IP address 20.0.0.2 to the port with number 80. It is possible to underline that, to express the *wildcards* feature, the number -1 is exploited to be compliant with the implementation of this feature in z3, which will be presented in Section 7.4.

```
<Property graph="0" name="ReachabilityProperty" src="10.0.0.-1"
    dst="20.0.0.2" dst_port="80" />
```

Listing 7.1. XML example of a medium-level reachability requirement

Listing 7.2 states an isolation requirement according to which the node with the IP address 10.1.1.1 must not be allowed to contact any node in the subnetwork 130.192.0.0/16 listening to port 80, if the source node is using any number port between 2000 and 3000.

```
<Property graph="0" name="IsolationProperty" src="10.1.1.1"
    dst="130.192.-1-.-1" src_port="[2000-3000]" dst_port="80"/>
```

Listing 7.2. XML example of a medium-level isolation requirement

7.4 Wildcards

The *wildcards* feature, introduced in the framework to model netmasks, is exploited in different components of the *ADP* module, from the implementation of the forwarding rules, to the definition of the Network Security Requirements and the firewall Filtering Policies. The main reason for its adoption has been that it can make much more powerful all the formulas where this feature is exploited.

Firstly, since the feature was already present in the old version of VEREFOO, the problems related to the previous implementation of this feature are presented; then, solutions to solve the incorrect logic formulations and to provide a more efficient management are provided.

7.4.1 Wildcards original idea and implementation

In the VEREFOO framework, wildcards are used to represent both an IP address and its netmask within a unique data structure; the goal is to minimize the number of variables which the z3 optimizer engine will receive as input, since each IP address itself, modelled as a *Datatype* structure, is composed by four variables, each one an integer representing a component of the address. In particular, to express that a part of an IP address is not related to a single host but a subnetwork, the conventional symbol $*$ is used for that part; to provide an example, the network 10.0.1.0/24 or 10.0.1.0 255.255.255.0 is modelled in VEREFOO as 10.0.1.$*$. In the z3 language, since it is not possible to represent the symbol $*$, it is substituted by the conventional value of -1.

Wildcards are a powerful tool which, in their original idea, would lead to minimize the number of rules *z3Opt* would evaluate as possible during the process of firewall auto-configuration, because they can directly merge single rules related to separated Network Security Requirements in a more limited number. The trade-off between the impossibility to express netmasks different from /24, /16, /8 and /0, being limited to a classful addressing scenario, and the potential future performance improvements was considered fair enough to introduce this feature in the framework.

However, the problem which initially arose was that, despite the original previsions, the performance which could be achieved as a result was worse than expected; for this reason, the feature was temporarily disabled and a post-processing task had the duty to merge the single rules which the optimizer had again to separately compute, in order to show the service designer the optimal output he effectively expected. This was not anyway an acceptable solution, because wildcards are clearly an essential factor in the VEREFOO implementation and in correctly modelling a Service Graph in a real usage environment.

First of all, the wildcards implementation was inspected in order to find a way to refine them. They were used in several circumstances, such as comparison between two IP addresses, the matching between a packet IP header and an IP address in the forwarding of the packet and the firewall auto-configured rules.
To cite as an example the first instance, the old implementation of the comparison between two IP addresses, called *ip1* and *ip2*, was realized as showed in the following formula:

$$\forall i \in \{1,2,3,4\} \,.\, ip1_i == ip2_i \vee ip1_i == * \vee ip2_i == * \implies ip1 == ip2 \quad (7.1)$$

What can immediately be noticed is that this implementation considers the four components of an IP address with wildcards, or in other words the four components of a netmask, as parallel and independent one from each other, despite it not being true. Actually, in a binary netmask, when a bit is set to 0 all the following ones must be 0 too and, in the same way, in VEREFOO if an IP address contains the value $*$ for one of its components, also the right-side elements should be $*$. Considering on the other hand the four components as parallel, IP addresses like 10.0.$*$.3 were created, having as consequence the need of a pre-processing task but particularly increasing by a big factor the space of solutions investigated by *Z3Opt*, despite that most of them were clearly incorrect without needing an analysis made by such a powerful but slow tool.

7.4.2 Wildcards new implementation

To solve the problem presented in Subsection 7.4.1, alternative formulas have been introduced to represent a hierarchical structure of an IP address with wildcards for all the scenarios on which they are used.

Considering again the the comparison between two IP addresses, called *ip1* and *ip2*, to provide an example of the new implementation of this feature, the correspondent new formula is the following:

$$\begin{aligned}
&(\forall i \in \{1,2,3,4\} \,.\, ip1_i == *) \vee (\forall i \in \{1,2,3,4\} \,.\, ip2_i == *) \vee \\
&(ip1_1 == ip2_1 \wedge ((\forall i \in \{2,3,4\} \,.\, ip1_i == *) \vee (\forall i \in \{2,3,4\} \,.\, ip2_i == *))) \vee \\
&((\forall i \in \{1,2\} \,.\, ip1_i == ip2_i) \wedge ((\forall i \in \{3,4\} \,.\, ip1_i == *) \vee (\forall i \in \{3,4\} \,.\, ip2_i == *))) \vee \\
&((\forall i \in \{1,2,3\} \,.\, ip1_i == ip2_i) \wedge ((ip1_4 == *) \vee (ip4_i == *))) \vee \\
&(\forall i \in \{1,2,3,4\} \,.\, ip1_i == ip2_i) \implies ip1 == ip2
\end{aligned} \quad (7.2)$$

In this formula, the cases which are considered as possible are basically two:

1. the two IP addresses do not have any $*$ values and in this case they must be completely equal;
2. one of the two IP addresses have a $*$ value; then it must have the right components as $*$ and the left components equal to the other IP address.

Hard constraints to further remark this hierarchy have also been enforced, so that enabling again wildcards has been possible.

Consequences of the introduction of these formulas and the rehabilitation of the usage of wildcards have been several and all of them important to achieve better performance in some algorithms:

- there are less combinations evaluated as possible by *z3Opt* during the hierarchical construction of each *Datatype* variable representing an IP address with wildcards;
- it is possible to specify as input end point in the Allocation Graph or in the Service Graph for the framework not only single clients, but also subnetworks (e.g. 10.0.1.$*$) since now the firewall rules can match correctly with this kind of addresses;
- there is no need any more of the post-processing task which was previously exploited and implemented by *Z3Translator* class in the original framework, where the single firewall rules were merged, because *z3Opt* now tries to configure the minimum number of rules in each firewall and therefore it directly tries to use wildcards. This aspect is fundamental for the firewall autoconfiguration algorithms because it will let the framework work with a reduced solution space.

7.4.3 Wildcards management

To manage wildcards, all the possible IP addresses with wildcards should be generated from all the IP addresses which are configured to the node of the Allocation Graph or of the Service Graph in input to the framework, before applying any heuristic and algorithm for the allocation of the Network Security Functions on the Allocation Places and their configuration.

For this purpose, the *WildcardManager* class has been introduced to the framework. At the beginning of the application, it builds all the possible address ranges from each distinct IP address existing on the Allocation or Service Graph, classifying them in four different levels according to the number of the position of the first -1 element – as the symbol $*$ is represented in the z3 language – from left-side; for instance, an address range like 10.-1.-1.-1 is of level two, while 10.0.0.-1 is of level four. After making this division, the calculated information is stored in four separated maps, where the key is represented by the address range with wildcards, while the value is the list of all the correspondent single IP addresses associated

to it; this way their retrieval can be done efficiently in terms of computation time, every time a Network Security Function should be configured with rules regarding IP addresses.

The most important method of the class is the *areAggregable* method, which receives two sets of IP addresses and returns a true boolean value if the first set can be aggregated in one larger address range with wildcards which does not include any IP address present in the second set; this feature is fundamental to implement the pruning algorithm through which auto-configuration of packet filters is performed, as it will be explained in Section 8.6.1 in more details.

7.5 Isolation Requirement

In VEREFOO, an isolation requirement is used to express the constraint for which a specific kind of communication from a source to a destination must be prohibited; in this thesis work, it basically indicates that the packets of this communication should be dropped by a packet filter according to the values of the IP quintuple.

Before presenting the constraints which are defined in the MaxSMT problem for an isolation requirement, the *match* function is introduced. Given a packet pk_0 and a security requirement r, the packet satisfies this requirement if each element of the requirement itself corresponds to the packet field, considering anyway that some elements can be a number or an interval. The behaviour of the *match* function is showed in Formula 7.3.

$$\begin{aligned} r.match(pk_0) \Leftrightarrow pk_0.origin \subseteq r.IPSrc \ \wedge \ pk_0.IPDst \subseteq r.IPDst \wedge \\ pk_0.portSrc \subseteq r.portSrc \ \wedge \ pk_0.portDst \subseteq r.portDst \wedge \\ pk_0.transportProto \subseteq r.transportProto \end{aligned} \tag{7.3}$$

Another function which must be introduced is the *addr* function; when it is applied on any node of the Service Graph or Allocation Graph, it returns the IP address with which this node is configured. This function is, consequently, exploited when defining the constraints for a Network Security Requirements in the creation of the packet by the source or in the verification that the packet has been correctly received or not by the destination.

Given these assumptions, considering $r \in R$ a specific isolation requirement, the constraints which must be defined to guarantee its satisfiability are expressed by means of the following Formulas 7.4, 7.5, 7.6, which are the results of the remodelling phase performed during this thesis work.

$$\exists e_i, e_j \in E_A.\ addr(e_i) == r.IPSrc \wedge addr(e_j) == r.IPDst \tag{7.4}$$

$$\forall k : n_k \in N_A \wedge l_{ik} \in L_A.\ \exists pk_0.\ send(e_i, n_k, pk_0) \wedge r.match(pk_0) \tag{7.5}$$

$$\begin{aligned} \forall k : n_k \in N_A \wedge l_{kj} \in L_A.\ \forall pk_0.\ (\ recv(n_k, e_j, pk_0) \wedge \\ pk_0.IPDst = addr(e_j) \implies \neg\, r.match(pk_0)) \end{aligned} \tag{7.6}$$

Only a combination of all the three formulas can lead to a correct verification of an isolation requirement, since each one has a precise purpose:

1. Formula 7.4 states that in the network a node with the same IP address of the source must exist and the same can be stated for the destination, because otherwise it is obvious that the packet will not ever be sent by a non-existent source or will never reach a non-existent destination node.

2. Formula 7.5 states that the source node of the isolation property should send to each one of its *firstHops* a packet whose destination coincide with the destination of the requirement itself, so that potentially all the possible paths inside the network can be exploited;

3. Formula 7.6 states that, if the destination node has received a packet from any of its *lastHops*, then the origin of the packets must not be the source of the requirement itself because otherwise it would mean that the source succeeded in reaching the destination.

Out of these formulas, the second one is essential to avoid some loops inside the network alongside with the forwarding rules which have been explained in Section 6.3; in fact, if the source node sent a packet to at least one of its neighbours and not to all of them, the optimizer would clearly decide to create only one packet, being able to reach the minimum cost for the optimization problem but using as a starting point a first-order logic formula which can be considered correct exclusively for a chain of nodes. The main problem in the previous implementation was, in fact, that not all the possible neighbours necessarily received a packet from the source of an isolation requirement and the verification of the constraint was consequently wrongly declared as satisfied by the MaxSMT solver.

It is worth mentioning that, in the implementation of the formulas in z3, the content of the maps *firstHops* of the source nodes and *lastHops* of the destination nodes are exploited to avoid the usage of quantifiers also in this context. This approach provided another contribution in improving the overall performance of the framework.

A complete example is presented in the following with Figure 7.1 to explain how these first-order logic formulas are sufficient and necessary to provide a complete verification of the satisfiability of an isolation requirement in a scenario where some loops between nodes are present, each end point is linked to more than one node and the requirements are all in the same direction.

In this example, the end point 10.0.0.1 is linked to the packet filters fw_1 and fw_2, the end point 10.0.0.2 is linked to the packet filters fw_1 and fw_2 while the end point 20.0.0.1 is linked to the packet filters fw_1 and fw_3; besides the three packet filters are linked creating a loop. The suppositions are that all the three packet filters work in *blacklisting* mode and that they cannot be removed by the specified logical topology because of some constraints which the service designer imposed, e.g. these firewalls are physical middleboxes which cannot be removed from the substrate network. For sake of simplicity, all the Network Security Requirements which will be considered in this example exclusively deal with IP addresses, supposing that the

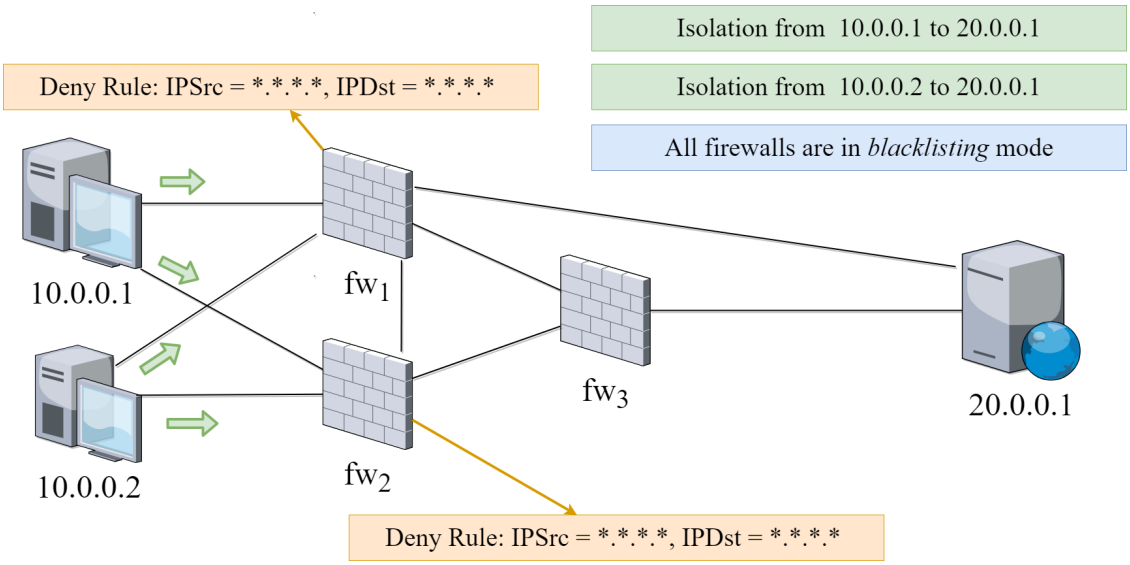


Figure 7.1. Example of isolation requirements

wildcards feature is exploited to represent both the ports and the transport-level protocol.

Two Network Security Requirements are specified in this scenario:

- isolation property from 10.0.0.1 to 20.0.0.1;
- isolation property from 10.0.0.2 to 20.0.0.1.

Formula 7.4 guarantees firstly that in the network the end points specified as sources and destinations of the two requirements – 10.0.0.1, 10.0.0.2, 20.0.0.1 – exist; this is necessary to model respectively the events of sending the packets by the source and receiving packets by the destination. About this aspect, it is possible to underline again that, if the service designer has specified a single isolation property from 10.0.0.∗ to 20.0.0.1, the constraints would have been the same because this requirement is automatically translated by a pre-processing task into the two aforementioned requirements.

Then, Formula 7.5 states that each source must send a packet, which matches the corresponding security requirement, to every neighbour through which a path towards the destination actually exists. This approach presents a new perspective in regard to the old design and implementation of the framework, where only loop-free chains were correctly analysed and for this reason any source would just send a packet to the network. However, if we suppose that both 10.0.0.1 and 10.0.0.2 would send a packet only to fw_1, then a single packet filter rule which blocks all the traffic distributed in fw_1 would be considered sufficient by z3 to satisfy the input requirements, when the result is actually incorrect because other paths exist from the sources to the final destination.
Instead, if they send a packet to both of their neighbours, at least two rules must be configured on the packet filters, for instance a rule on fw_1 and a second one on fw_2. This way all the possible paths are characterized by at least a blocking wall for the propagation of the packets.

Finally, Formula 7.6 states that, if the end point 20.0.0.1 receives a packet, then this packet must not match the security requirement, since the related communication must be blocked. For instance, if an end point 10.0.0.3 is added in the network with a link to fw_3, it can send a packet towards 20.0.0.1, which is acceptable because it does not contradict the specified set of security requirements.

7.6 Reachability Requirement

In VEREFOO, a reachability requirement is used to express the constraint for which a specific kind of communication from a source to a destination must be allowed; in this thesis work, it basically indicates that the packets of this communication should not be dropped by any intermediate packet filter according to the values of the IP quintuple.

Considering that the function *match* and *addr*, described in Section 7.5, are valid also for describing the model and implementation of a reachability requirement, given $r \in R$ a specific reachability requirement, the constraints which must be defined to guarantee its satisfiability are expressed by means of the following Formulas 7.7, 7.8, 7.9. In this case, differently from the isolation requirement, the formulas did not need a complete logic change, but they were reformulated to be compliant with the newly introduced model and to work with the new concepts of *firstHops* and *lastHops* to further increase the performance.

$$\exists e_i, e_j \in E_A.\ addr(e_i) == r.IPSrc \wedge addr(e_j) == r.IPDst \tag{7.7}$$

$$\exists k : n_k \in N_A \wedge l_{ik} \in L_A.\ \exists pk_0.\ send(e_i, n_k, pk_0) \wedge r.match(pk_0) \tag{7.8}$$

$$\begin{aligned} \exists k : n_k \in N_A \wedge l_{kj} \in L_A.\ \exists pk_0.\ recv(n_k, e_j, pk_0) \wedge \\ pk_0.IPDst = addr(e_j) \wedge\ r.match(p_0) \end{aligned} \tag{7.9}$$

Only a combination of all the three formulas can lead to a correct satisfiability and verification of a reachability requirement, since each one has a precise purpose:

1. Formula 7.7 states that in the network a node with the same IP address of the source must exist and the same can be stated for the destination, because otherwise it is obvious that the packet will not ever be sent by a non-existent source or will never reach a non-existent destination node;

2. Formula 7.8 states that the source node of the reachability requirement must send a packet which matches the requirement itself to at least one *firstHop*, not necessarily to every neighbour since the existence of one path is enough to satisfy the reachability requirement;

3. Formula 7.9 states that the destination node must receive a packet which matches the requirement itself from at least one of its *lastHops*, because it means that at least a path has been found.

It is important to underline that without Formula 7.9 there is the possibility that other nodes than the actual source of the requirement send a packet towards the same destination, while without Formula 7.8 there would be no proof that the packet sent by the correct source ever reached the destination. Then, all the considerations which were made about the *firstHops* and *lastHops* made for the isolation case are still valid also for the reachability requirements.

A complete example is presented in the following with Figure 7.2 to explain how these first-order logic formulas are sufficient and necessary to provide a complete verification of the satisfiability of a reachability requirement in a scenario where some loops between nodes are present and each end point is linked to more than one node.

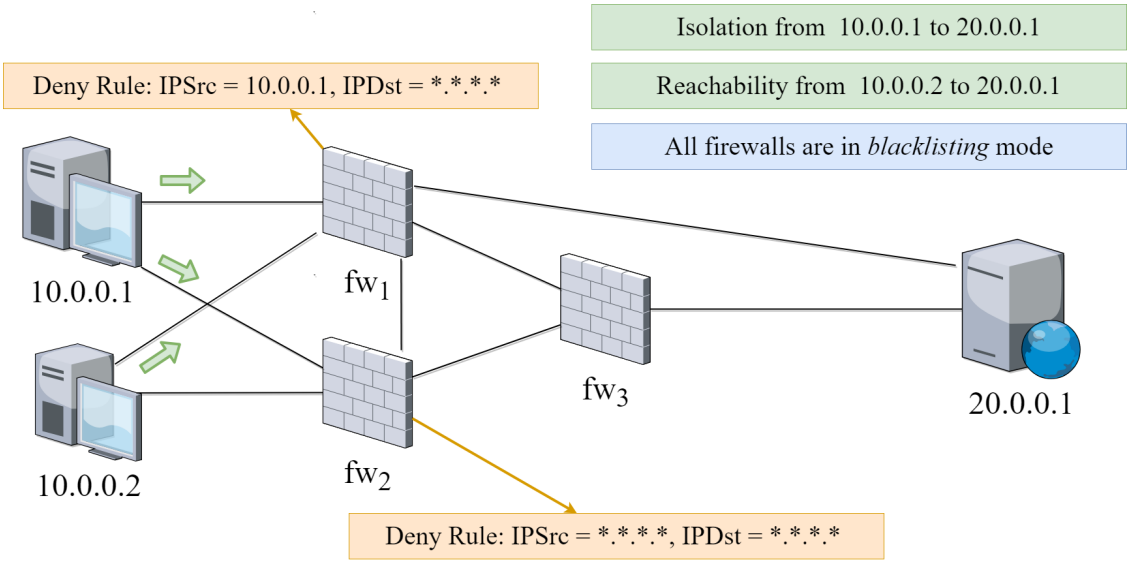


Figure 7.2. Example of reachability requirement

In this example, the end point 10.0.0.1 is linked to the packet filters fw_1 and fw_2, the end point 10.0.0.2 is linked to the packet filters fw_1 and fw_2 while the end point 20.0.0.1 is linked to the packet filters fw_1 and fw_3; besides, the three packet filters are linked creating a loop. The suppositions are that all the three packet filters work in *blacklisting* mode and that they cannot be removed by the specified logical topology because of some constraints which the service designer imposed, e.g. these firewalls are physical middleboxes which cannot be removed from the substrate network. For sake of simplicity, all the Network Security Requirements which will be considered in this example exclusively deal with IP addresses, supposing that the *wildcards* features is exploited to represent both the ports and the transport-level protocol; besides, they are specified for kinds of traffic which flow in the same direction.

Two Network Security Requirements are specified in this scenario:

- isolation property from 10.0.0.1 to 20.0.0.1;
- reachability property from 10.0.0.2 to 20.0.0.1.

Formula 7.7 guarantees firstly that in the network the end points specified as source and destination of the security requirements – 10.0.0.1, 10.0.0.2, 20.0.0.1 – exist; this is necessary to model respectively the events of sending the packets by the source and receiving packets by the destination.

Formula 7.8 states that the source must send a packet which matches the requirement to at least one *firstHop* which opens a path towards the destination. This approach derives from the application of the *De Morgan Law* to Formula 7.5: if all the paths must be considered between source and destination to prove the isolation, on the other hand to prove a reachability it is sufficient to demonstrate that at least one packet which matches the requirement has successfully reached the destination. Consequently, if 10.0.0.2 sends a packet only to fw_1, since its configured rule explicitly blocks only packets coming from 10.0.0.1, it is allowed to cross this packet filter and has two different paths to reach the target destination. A packet sent to fw_2 would instead be blocked; forcing the model to create this additional sending event would not be a mistake because the result would be correct, but it would decrease the overall performance of the framework since more packets would transit in the network and the set of constraints to satisfy the input requirements would not be any more the minimum set.

Finally, Formula 7.6 states that the end point 20.0.0.1 must receive at least a packet which matches the corresponding reachability requirement. In addition, in this formula there are not either any constraints about the maximum number of received packets which satisfy this condition or about packets with a different origin, because these aspects are not influential in the satisfaction of the reachability requirement.

7.7 Identity of the end points of Network Security Requirements

In the previous versions of the framework, the reachability and isolation requirements could be specified only between a client as source and a server as a destination; this aspect was, however, a deep limitation, since network functions like packet filters could just be used to protect servers, losing a consistent part of their real capabilities.

Thanks to the flexibility of the Allocation Graph and the new data structures created in the core of the framework, described previously in Section 6.2, it was possible to create an extension so that any node of the network can be specified as a source or a destination of a security requirement, allowing as a consequence interactions from client to client, from server to server and also from server to client. This was reached by creating all the possible loop-free paths inside the Allocation Graph not from clients to servers, but from sources to destinations of the Network Security Requirements.

The two classes of the framework which got the most significant modifications for this purpose are:

- *LinkCreator* class, which creates all the possible paths for every pair made by a source and the corresponding destination;
- *LinkProvider* class, which allows to efficiently retrieve the created paths.

Moreover, every time a recursive visit of the graph is needed these paths are considered instead of the original chains, so that there is not any need to build them more than once.

In the following part of this section, an example is provided by means of Figure 7.3 to describe the extensions which have been defined.

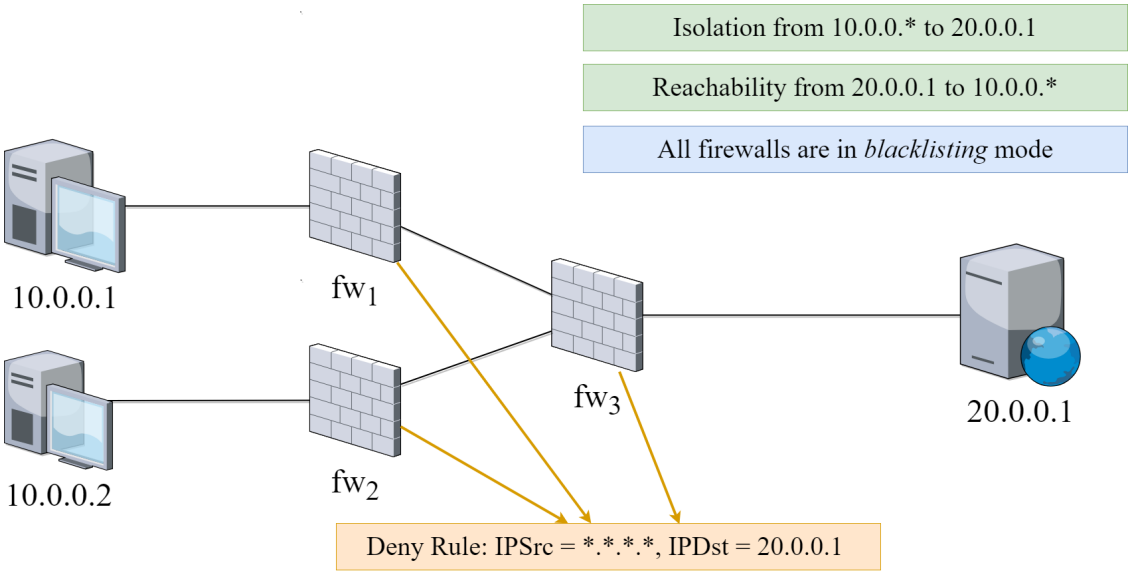


Figure 7.3. Example of a security requirement from server to client

In this scenario, 10.0.0.1 and 10.0.0.2 are two web clients, 20.0.0.1 is a web server and the intermediate middleboxes are three firewalls in *blacklisting* mode, which are in fixed positions according to some constraints set by the service designer and cannot be removed by the framework. The two Network Security Requirements which the user specifies as input are:

- isolation property from every client belonging to the network 10.0.0.0/24 to the server 20.0.0.1;
- reachability property from the server 20.0.0.1 to every client belonging to the network 10.0.0.0/24.

In the past, the z3 model would not have been able to reach a valid solution; after the modifications, instead, the auto-configuration of the firewalls must consider also the packets sent by the server to the clients. The most interesting consideration is that in this example the communication is basically bidirectional, because the packets sent by the server to a client follow the same path, in the opposite sense, as

the packets created by the client; as a consequence, it becomes more burdensome for the optimizer to find the optimal solution.

In this case, the result which is achieved is that, in all the three firewalls, a single blocking rule is configured, whose source is the full route 0.0.0.0/0, modelled by means of the *wildcards* as ∗.∗.∗.∗, and whose destination is the server 30.0.0.1. This rule is sufficient to block all the traffic coming to the clients and, since its default behaviour is the blacklisting, the packets sent by the servers are not dropped but they can reach their targets.

7.8 Multiple Network Security Requirements between the same pair of end points

In the management of real networks, it is common that more than a single Network Security Requirement involve the same pair of end points. For example, when a server offers multiple services on different transport-level port numbers, some clients could be allowed to contact it only for a specific service, while they could be prohibited to request the others; this traditional scenario is represented in Figure 7.4, where the client 10.0.0.1 must be able to reach the server 20.0.0.1 at the destination port 80, while it must not contact it at the destination port 90.

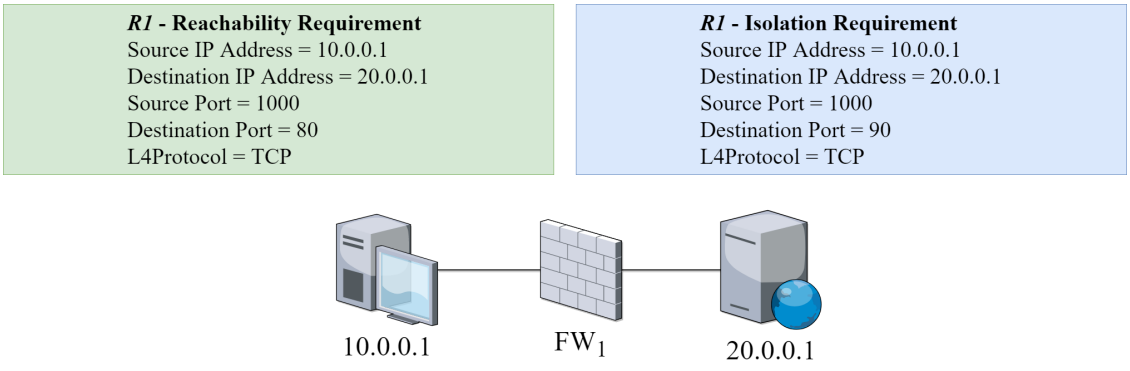


Figure 7.4. Example of multiple requirements between the same pair of end points

It is worth underlining, in particular, that the two Network Security Requirements defined in this example share all the parameters of the IP quintuple except for a single one, the transport-level destination port; consequently, the firewall FW_1 which is allocated between the two end points, under the hypothesis of a whitelisting configuration mode, should define a specific rule to allow the traffic which targets the destination port 80.

However, in the old z3 model for the representation of the Network Security Requirements, the presence of some hard constraints forced each traffic source to send packets with the same values of the source port, destination port and transport-level protocol. Consequently, to manage the possibility of multiple security requirements between the same pair of end points, a pre-processing task was introduced, to multiply the source by a factor which was equal to the number of security requirements

where the same destination was involved. This way, each one of these artificial sources could send packets with different ports and transport-level protocols.

This approach is, nevertheless, characterized by several limitations. First of all, in terms of performance and scalability, wildcards and pruning strategies cannot be used any more to reduce the firewall rules; in fact, each artificial source – called *virtual node* – is characterized by a name which cannot be divided into the traditional four elements of an IP address. Moreover, since the total number of nodes and links in the network increases, then also the solution space of the z3 model increases with the introduction of additional constraints.

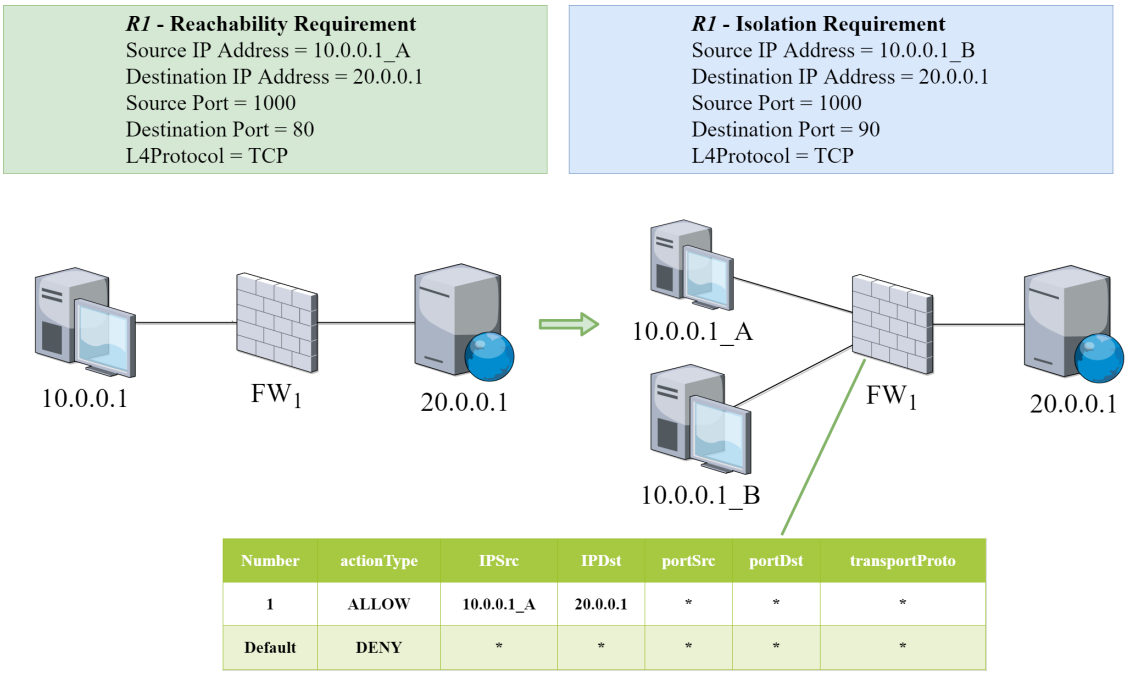


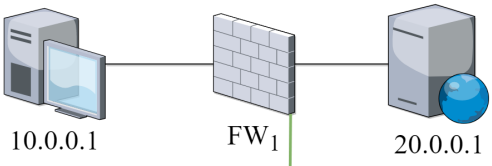
Figure 7.5. Firewall Policy result in the old model

But, among all the drawbacks of this approach, the most critical one is a lacking adherence of the model to the reality. Considering the previous example, Figure 7.5 shows how the policy for firewall FW_1 was configured after the execution of the framework. After the client 10.0.0.1 is split in two different end points, each one is then characterized by a different name (10.0.0.1_A and 10.0.0.1_B), that not only are not IP addresses, but are actually different strings. Consequently, the firewall does not interpret these virtual nodes as a single one and, when the policy must be decided, a specific rule is simply configured to allow the traffic coming from 10.0.0.1_A, despite it not existing in reality. The achieved result is correct for the pre-processed graph, but it is not adherent to the real network.

After the introduction of the formulas illustrated in Sections 7.5 and 7.6 to model the isolation and reachability properties, the result which is achieved by means of a simulation of the framework is different, as it is showed in Figure 7.6. Now, instead of splitting the source of the two Network Security Requirements, the client 10.0.0.1 is simply allowed to send packets with different ports and transport-level protocols. The firewall, consequently, is automatically configured with a single

***R1* - Reachability Requirement**
Source IP Address = 10.0.0.1
Destination IP Address = 20.0.0.1
Source Port = 1000
Destination Port = 80
L4Protocol = TCP

***R1* - Isolation Requirement**
Source IP Address = 10.0.0.1
Destination IP Address = 20.0.0.1
Source Port = 1000
Destination Port = 90
L4Protocol = TCP



Number	actionType	IPSrc	IPDst	portSrc	portDst	transportProto
1	ALLOW	*	*	*	80	*
Default	DENY	*	*	*	*	*

Figure 7.6. Firewall Policy result in the new model

rule, which allows the packets with 80 as destination port to pass and blocks all the others.

The same concept can be applied in scenarios where the multiple Network Security Requirements, defined for the same pair of end points, differ for other parameters, such as the source port or the transport-level protocol. This way, richer scenarios can be introduced in the framework to get the optimal firewall allocation and configuration, the resulting model is compliant with the IP addresses of the nodes in the input Service Graph and the filtering policy rules exploit all the features which have been designed for all the elements of the IP quintuple.

Chapter 8

Packet Filter Firewall

The automatic allocation and auto-configuration of packet filter firewalls represent a central component of this thesis work; this aspect is built on all the newly introduced features, like the Allocation Graph, and the remodelling of other elements such as the Network Security Requirements. It is also a central aspect of the *Security Automation* approach representing the foundation of VEREFOO.

After Section 8.1 provides a brief description about the capabilities of a packet filter, Section 8.2 defines the structure of the Filtering Policy of a firewall which has been defined, whereas Section 8.3 illustrates the XML elements which the thesis inherited from the previous framework and exploited to represent the modelled Filtering Policies.

Then, the remainder of this chapter focuses on describing in Section 8.6 how the automatic allocation feature has been introduced in the *ADP* module, while in 8.6.1 how the constraints related to the auto-configuration feature of the Filtering Policies have been remodelled to achieve better performance and to exploit the *wildcards* feature.

Finally, in Section 8.7 a complete clarification example is illustrated, showing how the framework is able to compute the optimal allocation schema and configuration of virtual firewalls starting from a Service Graph and a set of Network Security Requirements; the purpose of this final section is to show a complete workflow and, at the same time, how the goals related to the soft constraints are effectively achieved.

8.1 Introduction to Packet Filter Firewall

The packet filter is a technology of firewall which is able to make a decision about forwarding each incoming packet basing it exclusively on IP addresses and ports of both source and destination, information collocated respectively in the layer three (network) and four (transport) of the ISO/OSI stack.

Its main advantages are that it is easy to implement, it is application independent and overall its performance is very good because it is able to avoid incoming

Denial-Of-Service attacks processing the packets in the first levels of the networking stack of the Operating System.

On the other hand, its configuration is not trivial, because rules regarding application layer, TCP flux and UDP datagrams should be translated in rules which regards only IP addresses and ports; besides, the provided security level is low, because decisions are based on a restricted basin of information. In addition, if only a packet filter is used as security control, it is difficult to authenticate users and to support services like FTP based on a dynamic allocation of ports.

Despite its flows, a packet filter is one of the most important and used Network Security Function inside a network, because the screening router on which it is installed can easily drop most of the packet belonging to prohibited communication, unburdening more complex appliances like an Application-Level Gateway or a Deep Packet Inspector.

8.2 Model of the Filtering Policy of a firewall

Each firewall is characterized by a *Filtering Policy* representing the configuration according to which this Network Security Function decides if a received packet must be discarded or should be forwarded to the out-ports which allow to reach its destination. The Filtering Policy can be established by the service designer for a firewall which he decides to allocate in a specific position of the input Service Graph, if he has sufficient knowledge in the security field or if this policy comes as a result of a previous run of the framework; otherwise, by default for each firewall which is automatically allocated on the Allocation Graph the *ADP* module provides a complete configuration.

Remembering from Subsection 6.2.2 that P_A is the set of the Allocation Places of an Allocation Graph, given $p_k \in P_A$ an Allocation Place where a firewall can be allocated, the related Filtering Policy is characterized by two components:

- a default action δ_k;
- a set of policy rules Ψ_k, which establish that a packet which matches their conditions should be managed with the corresponding actions. The assumption is that all the policy rules are not redundant between themselves.

When a packet is received by a packet filter, firstly the conditions of each policy rule in the Ψ_k set are applied to the field of the packets; if the matching between a rule and the packet is positive, then the corresponding action – which can be *ALLOW* or *DENY* – decides if the packet can be forwarded or must be discarded. If no rule matches the fields of the packets, it is managed according to the default action δ_k, which has less priority then all the other rules.

The default action δ_k can be assigned two different values:

1. $\delta_k = DENY$, if the firewall configuration is whitelisting;
2. $\delta_k = ALLOW$, if the firewall configuration is blacklisting.

Each rule in Ψ_k is, instead, characterized by the following model:

[actionType - IPSrc - IPDst - portSrc - portDst - transportProto]

where:

- *actionType* can have the values *ALLOW* or *DENY* and specifies which action must be performed on each packet which satisfies the conditions of the rule;
- *IPSrc* is the source IP address of the communication for which the action is applied;
- *IPDst* is the destination IP address of the communication for which the action is applied;
- *portSrc* is the transport-level source port of the communication for which the action is applied;
- *portDst* is the transport-level destination port of the communication for which the action is applied;
- *transportProto* is the transport-level protocol of the communication for which the action is applied.

8.3 Implementation in the XML schema of the Filtering Policy of a firewall

A packet filter is represented in the XML schema by a *node* element whose *functional_type* attribute has *FIREWALL* as value. In addition to the traditional internal elements of a *node*, such as the set of *neighbours* elements, a packet filter is also characterized by a specific configuration which contains a *firewall* element.

A *firewall* element has a *defaultAction* attribute, which can be assigned the value *ALLOW* or *DENY* depending on the configuration mode – respectively, blacklisting or whitelisting – and a set of *elements*, representing the Filtering Policy rules and expressed by means of a medium-level representation.

Each Filtering Policy *element* is characterized by the following attributes:

- the *action* element, which expresses the rule action (*ALLOW* or *DENY*);
- the *source* element, which expresses the source IP address of the rule (it can also be the corresponding unique identifier or a larger address range which corresponds to a subnetwork including more than just one host);
- the *destination* element, which expresses the destination IP address of the policy (it can also be the corresponding unique identifier or a larger address range which corresponds to a subnetwork including more than just one end point);

- the *protocol* element, which expresses the transport-level protocol (TCP, UDP or any);
- the *src_port* element, which expresses the transport-level source port (it must be a number between 0 and 65535, a range or it can be empty if the rule is only specified for IP addresses);
- the *dst_port* element, which expresses the transport-level destination port (it must be a number between 0 and 65535, a range or it can be empty if the rule is only specified for IP addresses);
- the *directional* element, which expresses if the rule is valid for both the directions or if it is valid only for the specified direction.

Several combinations can be made from these six information pieces, especially specifying address ranges and port intervals, so that richer rule expressions can be defined.

The model of a single Filtering Policy rule which is defined in the XML schema is showed in Listing 8.1.

```
<xsd:element name="elements">
       <xsd:complexType>
              <xsd:sequence>
                     <xsd:element name="action" type="ActionTypes"
                         minOccurs="0" default="DENY"/>
                     <xsd:element name="source" type="xsd:string"/>
                     <xsd:element name="destination" type="xsd:string" />
                     <xsd:element name="protocol" type="L4ProtocolTypes"
                         minOccurs="0" default="ANY"/>
                     <xsd:element name="src_port" type="xsd:string"
                         minOccurs="0"/>
                     <xsd:element name="dst_port" type="xsd:string"
                         minOccurs="0"/>
                     <xsd:element name="directional" type="xsd:boolean"
                         minOccurs="0" default="true"/>
              </xsd:sequence>
       </xsd:complexType>
</xsd:element>
```

Listing 8.1. XML schema for the Filtering Policy

An example of Filtering Policy which the framework could automatically generate is, furthermore, showed in Listing 8.2.

```
<node name="10.0.0.2" functional_type="FIREWALL" >
       <neighbour name="10.0.0.1" />
       <neighbour name="10.0.0.4" />
       <configuration description="Firewall_configuration"
           name="Configuration01">
              <firewall defaultAction="DENY">
                     <elements>
                            <action>ALLOW</action>
                            <source>10.0.0.1</source>
                            <destination>10.0.0.4</destination>
                            <protocol>ANY</protocol>
```

```
                                <src_port>*</src_port>
                                <dst_port>80</dst_port>
                            </elements>
                        </firewall>
                    </configuration>
                </node>
```

Listing 8.2. XML example of a Filtering Policy

8.4 Objectives of the MaxSMT problem

The two main objectives of the MaxSMT problem, in relation to the firewalls which can protect the Service Graph from some cybersecurity attacks, are the following two:

1. the minimization of the number of firewalls which are allocated in the Allocation Graph, so that the resource consumption will be minimized when the VNFs implementing the packet filters are deployed on a substrate network, independently from the actual requirements of each virtual instance (e.g. RAM, CPU) which are not considered in the defined approach because already managed by another existent part of the *ADP* module;

2. the minimization of the number of rules for each firewall Filtering Policy, so that the resulting configuration is more easily read by the service designer and can be managed or modified with less risks in making the typical human errors; additionally, the minimum set of rules would also require the minimum amount of memory to store them in the corresponding virtual function, when deployed. The most important motivation of this objective is however that, with less rules, the filtering operations are faster, because they have to parse a shorter list.

These two objectives are independent from a theoretical point of view, but they are modelled by means of shared variables in the z3 model of the MaxSMT problem, so that the optimal solutions are achieved for both of them contemporary. However, as it will be clearer after all the first-order logic formulas will have been presented, the priority is assigned to the minimization of the number of allocated firewalls, since deploying an additional function is much more expensive than adding a new rule in an existing firewall instance.

8.5 Automatic Allocation of Firewalls

The automatic definition of the allocation schema of the firewalls on the Allocation Graph, given a set of Network Security Requirements, represents a fundamental novelty introduced by this thesis work in the *ADP* module. In the past, the framework was actually able to manage with not good performance only the creation

of the auto-configuration of static firewalls, whose position was established beforehand, and at the end of the execution it could remove some optional firewalls if they were proved useless by the optimizer.

The newly proposed approach, on the other hand, works on an Allocation Graph which comes from a Service Graph, so the service designer has not to manually define all the places where a firewall can be allocated, unless he wants to perform this operations by means of a set of additional constraints, already described in Subsection 6.2.2. This is fundamental because the actual optimal solution can be reached only when the complete space of possible solutions in terms of allocation is exhaustively explored.

Since the optimal solution would be that there is no need to instantiate virtual functions for the packet filtering operation, then for each Allocation Place a single soft constraint is defined to express the preference for which no firewall is allocated on the Allocation Graph. The complete set of these soft clauses is represented in Formula 8.1, where c_k is the weight assigned to the corresponding soft clause and which participates to the MaxSMT objective sum to maximize, if the clause itself is satisfied.

$$\forall k : p_k \in P_A.\ Soft(allocated(p_k) = false,\ c_k) \tag{8.1}$$

An indication about the value which is assigned to the c_k cost will be provided at the end of Section 8.6, because c_k depends on the costs assigned to the soft constraints related to the auto-configuration of the Filtering Policy. However, a *thumb rule* which can give an idea about the priority criteria defined in this MaxSMT problem is that this cost must be bigger than the sum of the costs of every other soft clauses related to the same Allocation Place. This choice is critical not only to reach the real objectives, but also in terms of performance, because when the optimizer engine establishes that a firewall must not be allocated on a specific Allocation Place, then it does not make sense to try to satisfy all the soft constraints related to the definition of the policy rules, because they would be anyway never satisfied.

8.6 Automatic Configuration of the Filtering Policies

After receiving an input R set of Network Security Requirements, the *ADP* module must establish the minimum set of Filtering Policy rules for each allocated firewall. Given as assumption that the the default action δ_k of each firewall which can theoretically allocated on a specific Allocation Place is established by the service designer – *DENY* if he prefers to have a higher level of security, *ALLOW* if he prefers to keep the configuration of the firewall as much simple as possible –, the critical problem is represented by the identification of how many rules in the worst case a firewall allocated in $p_k \in P_A$ could effectively need. For each one of these placeholder rules – from here on they are called placeholders because the decisions about their effective usefulness has still not been established –, the optimizer engine

would finally decide if they are needed and configure them if necessary; for this purpose, a set of hard and soft constraints are introduced in the MaxSMT problem instance for each rule. Consequently, it is essential to limit the cardinality of this set of placeholder rules.

A dummy approach – which was followed in the old version of this framework – was to introduce, for the Filtering Policy of each Allocation Place $p_k \in P_A$ of the Allocation Graph, a placeholder rule in Ψ_k for each input Network Security Requirement. In other words, if theoretically in the framework N Network Security Requirements are specified and F firewalls are deployed in the logical topology, then a trivial solution could be to create in the z3 model N equivalent rules for each packet filter, with a resulting total number of N*F rules. However, this methodology leads to an optimized process, because several security requirements do not actually need the configuration of a specific rule.

A central work performed during this thesis, consequently, has been to identify some pruning strategies by means of which it is possible to reduce the cardinality of the set of placeholder rules, identifying the only security requirements which could actually need a specific rule in the Filtering Policy of a firewall. Then, a second step has been to remodel the soft constraints through which the auto-configuration of each packet filter is achieved, so that a reduction of their number could lead the framework to achieve better performance.

8.6.1 Packet filter auto-configuration algorithms

The pruning strategies which have been developed have the purpose to create, for each $p_k \in P_A$, the minimum set R_k, that is the minimum set of the Network Security Requirements which effectively interest a firewall potentially allocated in this position and could require the configuration of a specific rule in the corresponding Filtering Policy. This step is critical for the performance, because the cardinality of this set is the size itself of the set of placeholder rules Ψ_k.

The first two principles which were identified to reach this goal are the following:

1. a firewall could be interested by a Network Security Requirement only if the packets related to this requirement pass through the Allocation Place on which the packet filter could be installed;

2. a specific rule could be needed in the Filtering Policy of a firewall to satisfy a Network Security Requirement only if this constraint is not already enforced by the default action of the packet filter itself – in other words, only if the requirement type (i.e. reachability or isolation) is of the opposite packet filter's default action, because for a whitelisting packet filter only requirements about reachability constraints should be considered since the others are automatically managed by the default action (and vice versa).

To exploit these principles, the following two functions can be introduced:

- *path(IPSrc, p_k, IPDst)* returns true if $p_k \in P_A$ is present in at least a path between the IP addresses specified as source and destination;

- *enforce*(δ_k, r) returns true if the requirement $r \in R$ is automatically enforced by the default rule δ_k of the firewall instance, so it does not require any specific firewall rule in Ψ_k.

Consequently, given the set of all the input Network Security Requirements R and an Allocation Place $p_k \in P_A$ with a specific default action δ_k, then the construction of the initial content of the R_k set is performed as showed in Formula 8.2, where both of the above introduced functions are exploited contemporary.

$$\begin{aligned} \forall r \in R.\, \forall p_k \in P_A.\, (\, & path(r.IPSrc, p_k, r.IPDst) \wedge \\ & \neg\, enforce(\delta_k, r) \implies r \in R_k) \end{aligned} \tag{8.2}$$

Moreover, a third principle has been later introduced to further reduce the cardinality of this R_k set by means of considerations about the possibility to directly merge some security requirements in a single one. For example, it is possible to suppose that a blacklisting firewall is interested by two different isolation requirements where the first one is related to the communication from 10.0.0.1 to 20.0.0.1, while the second one from 10.0.0.2 to 20.0.0.1, and that no other requirements interest this firewall; then, by exploiting the *wildcards* feature, rehabilitated in the framework after the modification performed during this thesis work, the packet filter would actually need a single rule to manage the traffic from 10.0.0.$*$ to 20.0.0.1, instead of two. For this purpose, if this scenario is identified before introducing the constraints in the z3 model of the MaxSMT problem, it is possible to further reduce the cardinality of the R_k set and, consequently, the number of placeholder rules in Ψ_k.

On the other hand, if in the aforementioned example a third requirement is introduced – in particular, a reachability requirement from 10.0.0.3 to 20.0.0.1 – , even if this constraint is already managed by the default action, since the rule which blocks all the traffic from 10.0.0.$*$ to 20.0.0.1 has higher priority, it would block also the packets coming from 10.0.0.3. Consequently, it has been fundamental to identify only the cases where this cardinality reduction can be applied to the R_k set.

According to the working mode of a firewall – blacklisting or whitelisting –, this third principle is structured in a different way; both cases are presented in the following, to help the understanding of how this algorithm works.

In case of a blacklisting firewall, if:

1. a subset of isolation requirements in R_k have the same destination,

2. in these isolation requirements the sources can be grouped in a larger IP address range (e.g. 10.0.$*$.$*$, $*$.$*$.$*$.$*$),

3. the other isolation and reachability requirements with the same destination and whose packets pass through the packet filter do not have sources included in that address range

then the packet filter can have only one *DENY* placeholder rule for all this subset of isolation requirements.

Dually, in case of a whitelisting firewall, if:

1. a subset of reachability requirements in R_k has the same destination,
2. in those reachability requirements the sources can be grouped in a larger IP address range (e.g. 10.0.∗.∗, ∗.∗.∗.∗),
3. the other isolation and reachability requirements with the same destination and whose packets pass through the packet filter do not have sources included in that address range

then the packet filter can have only one *ALLOW* placeholder rule for all this subset of reachability requirements.

The same consideration can be applied when the algorithm tries to merge a set of Network Security Requirements which share the same source, instead of the same destination; in the presented algorithms, actually, it is sufficient to replace the term *source* with the term *destination* and vice versa.

The complete algorithm, which exploits all the three described principles, is implemented in the *PacketFilterManager* class, which decides if a packet filter is interested in a Network Security Requirement, verifies if the default action of the packet filter is orthogonal to the security type and overall minimizes the number of placeholder rules inside each firewall as much as possible, to reduce the final number of soft and hard constraints inside the z3 model.

Figure 8.1 can be used to provide an example and clarify how the developed packet filter auto-configuration algorithms work.

In this scenario, all the five packet filters are configured in blacklisting mode and three Network Security Requirements are stated:

1. an isolation requirement from 10.0.0.1 to 20.0.0.1;
2. an isolation requirement from 10.0.0.2 to 20.0.0.1;
3. a reachability requirement from 10.0.1.1 to 20.0.0.1.

Initially all packet filters have a rule for each Network Security Requirement, independently of the requirement type and the fact that packets belonging to the satisfaction of a requirement actually cross the firewalls.

Firstly, the first principle – based on the *path* function – establishes that the packet filters in p_1, p_2 and p_3 can configure only one rule instead of three, because respectively their interested requirements are exclusively *Requirement1* for the first, *Requirement2* for the second and *Requirement3* for the third. Instead, the firewall in p_4 could require two rules because it is crossed by packets related to *Requirement1* and *Requirement2*, while the firewall in p_5 could require three rules, directly one for each requirement.
This step is fundamental, because it provides the most consistent reduction of rules

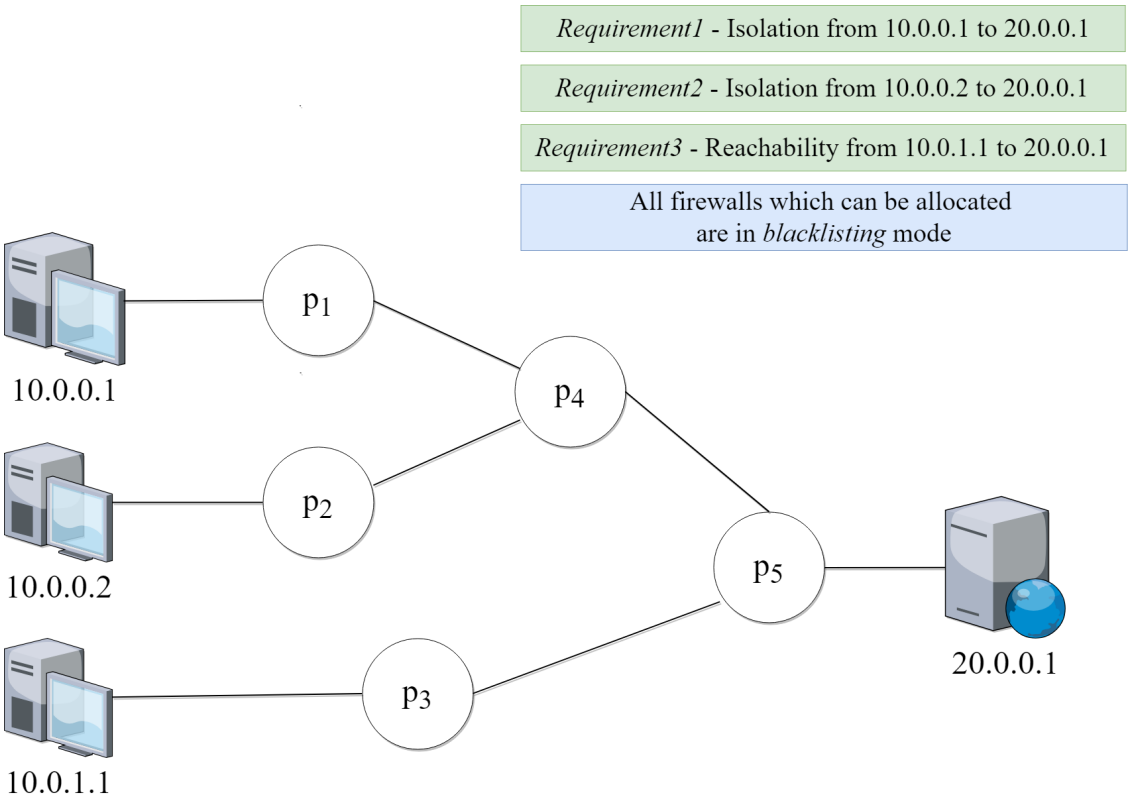


Figure 8.1. Example of packet filter auto-configuration algorithm

number despite being quite simple to implement through the recursive visit of the Allocation Graph.

Secondly, the second principle – based on the *enforce* function – establishes that the firewall in p_3 can avoid to configure any rule, because the only requirement which is of interest for this packet filter is *Requirement3*, a reachability property with an equivalent type of the default action of the blacklisting mode; for the same reason, the firewall in p_5 could require at most only two rules instead of three. On the other hand, this principle does not influence the number of placeholder rules of any other packet filter, for which the interested requirements are isolation properties.

Thirdly, the third principle – based on the *wildcards* features – establishes that both the firewalls in p_4 and p_5 require only one placeholder rule, because using wildcards it is sufficient to block packets coming from 10.0.0.1 and 10.0.0.2, since these IP addresses can be merged in the address range 10.0.0.0/24, equivalent to 10.0.0.$*$ in the framework formulation and configurable in the z3 model as 10.0.0.-1 by means of soft constraints.

8.6.2 Configuration of a packet filter in z3

After defining the cardinality of the R_k set, the next step has been to remodel the clauses of the MaxSMT problem exploited to auto-configure the needed Filtering Policy rules, since the previous model was not sufficiently optimized.

Inside the z3 model, a first hard constraint defines the default behaviour of the packet filter, according to the choice (blacklisting or whitelisting) made by the service designer if he decided to introduce some packet filters on the topology or if the input file comes from a previous run of the framework; otherwise, whitelisting working mode is the default choice of VEREFOO for the firewall Filtering Policies to achieve better security, despite it being more difficult to configure.

In particular, Formula 8.3 describes the hard constraint for a blacklisting configuration, while Formula 8.4 describes the hard constraint for a whitelisting configuration.

$$\delta_k = ALLOW \tag{8.3}$$

$$\delta_k = DENY \tag{8.4}$$

Then, in order to provide the auto-configuration feature of the specific rules, for each requirement in R_k which a packet filter must manage, a corresponding possible rule must be created inside the z3 model; then, the optimizer is in charge of deciding if that rule is needed and, in this case, to valorize its components.

First of all, the combination of the soft constraint represented by the Formula 8.5 and the hard constraint expressed in the Formula 8.6 is exploited to represent the ideal condition in which no firewall rule is configured. Basically, a rule is considered as not configured if the source and destination IP addresses are not set.

$$\forall \psi_i \in \Psi_k.\ Soft(\psi_i.IPSrc = \emptyset \wedge \psi_i.IPDst = \emptyset,\ c_{ki1}) \tag{8.5}$$

$$\forall \psi_i \in \Psi_k.\ \psi_i.IPSrc = \emptyset \wedge \psi_i.IPDst = \emptyset \implies \psi_i = \emptyset \tag{8.6}$$

In more details, Formula 8.5 states that the best situation, for each placeholder rule, is that the two main components – source and destination IP addresses – are not actually configured by the optimizer engine, whereas Formula 8.6 states that, if in a placeholder rule the source and destination IP addresses are not configured, then the rule itself is considered empty, i.e. not configured and useless.

In the original version of the framework, instead of the single soft constraint 8.5 for each rule, eight soft constraints were defined, each one for an element of an IP address. This way not only *z3Opt* had a large space of solutions to investigate because of the presence of eight separated soft constraints, but incorrect situations could also be reached as result because this problem deals with optimization. Firstly, an IP address like 10.0.∗.3 could be configured in a rule, since each one of the four components is considered as independent; instead, if a component is absent, also the others should be ∅. Secondly, if the source IP address is ∅, then also the destination must be the same, because otherwise only half rule is configured while the remaining part is empty.

If however at least a placeholder rule is configured, then the firewall must be allocated on the corresponding Allocation Place, because it means that this rule is needed to satisfy an input Network Security Requirement. Therefore, the hard

constraint represented in Formula 8.7 is introduced in the z3 model, in order to state that if at least one rule in the Filtering Policy is not empty, i.e. it is configured, then the firewall must be necessarily allocated.

$$\exists \psi_i \in \Psi_k : \ \psi_i \neq \emptyset \implies allocated(p_k) = true \tag{8.7}$$

If the soft constraint presented in 8.5 cannot be respected, the second ideal scenario would be that the placeholder rule which needs to be configured would use the *wildcards* feature in each component of the IP quintuple; this is achieved by means of additional eleven soft constraints, which contribute to a total of thirteen soft clauses for each placeholder rule, alongside with 8.5. This way, the possibility that this rule satisfies contemporary more than just one Network Security Requirement is higher. These soft constraints are presented by means of the following FOL formulas:

$$\forall \psi_i \in \Psi_k . \forall j \in \{1,2,3,4\} \, . \, Soft(\psi_i.IPSrc_j = *, \ c_{ki2j}) \tag{8.8}$$

$$\forall \psi_i \in \Psi_k . \forall j \in \{1,2,3,4\} \, . \, Soft(\psi_i.IPDst_j = *, \ c_{ki3j}) \tag{8.9}$$

$$\forall \psi_i \in \Psi_k . \, Soft(\psi_i.pSrc = [0, 65535], \ c_{ki4}) \tag{8.10}$$

$$\forall \psi_i \in \Psi_k . \, Soft(\psi_i.pDst = [0, 65535], \ c_{ki5}) \tag{8.11}$$

$$\forall \psi_i \in \Psi_k . \, Soft(\psi_i.tProto = *, \ c_{ki6}) \tag{8.12}$$

It is essential that the sum of the costs assigned to the soft constraints expressed by Formulas 8.8 and 8.9 is inferior than the cost assigned to the clause represented by Formula 8.5. In fact, the *wildcards* feature is considered less primary than the absence of configuration itself of the firewall rule. This way, the optimizer always evaluates as better the possibility to avoid the construction of the rule before trying to set the *wildcards* for its components.

$$\forall \psi_i \in \Psi_k . \sum_{j=1}^{4} (c_{ki2j} + c_{ki3j}) < c_{ki1} \tag{8.13}$$

Another constraint is related to the cost assigned to the soft constraint represented by Formula 8.1; since the ideal situation is that the firewall instance is not allocated, then this clause has a higher priority than the others.

$$\sum_{i:\psi_i \in \Psi_k} \left(c_{ki1} + \sum_{j=1}^{4} (c_{ki2j} + c_{ki3j}) \right) < c_k \tag{8.14}$$

Finally, the *behaviour* function, which is exploited by the forwarding rules described in Section 6.3, is built upon the configured Filtering Policy rules. Basically,

this function represents how the specific rules which could be configured on a firewall should match any received packet; it is clear that the construction of the firewall behaviour is, however, different accordingly to the default action of the packet filter itself and accordingly to the two different scenarios in which there are configured rules or no placeholder rule has been effectively valorized by the optimizer.

For this reason, four alternative ways are showed to define how this function works on a generic packet pk_0:

1. Formula 8.15 describes how *behaviour* is built when the firewall is in whitelisting working mode and at least a rule is configured;
2. Formula 8.16 describes how *behaviour* is built when the firewall is in whitelisting working mode and no rule is configured;
3. Formula 8.17 describes how *behaviour* is built when the firewall is in blacklisting working mode and at least a rule is configured;
4. Formula 8.18 describes how *behaviour* is built when the firewall is in blacklisting working mode and no rule is configured;

In these formulas, it is worth underlining that the notation $\Psi_k = \emptyset$ means that the optimizer engine has not configured any specific rule for the firewall in p_k.

$$\begin{aligned} \Psi_k \neq \emptyset \wedge \delta_k = DENY \implies \\ behaviour(p_k, pk_0) = \bigvee_i r_i.match(pk_0) \end{aligned} \tag{8.15}$$

$$\begin{aligned} \Psi_k = \emptyset \wedge \delta_k = DENY \implies \\ behaviour(p_k, pk_0) = false \end{aligned} \tag{8.16}$$

$$\begin{aligned} \Psi_k \neq \emptyset \wedge \delta_k = ALLOW \implies \\ behaviour(p_k, pk_0) = \neg \left(\bigvee_i r_i.match(pk_0) \right) \end{aligned} \tag{8.17}$$

$$\begin{aligned} \Psi_k = \emptyset \wedge \delta_k = ALLOW \implies \\ behaviour(p_k, pk_0) = true \end{aligned} \tag{8.18}$$

8.7 Clarification example about allocation and configuration of firewalls

The two objectives of the MaxSMT problem, i.e. the optimal allocation schema and configuration of the packet filter firewalls, are achieved with respect of a set of Network Security Requirements. A clarification example will be presented in the following, through a series of figures and consequent considerations, in order

to clearly show how the aforementioned objectives are achieved in the methodology presented in this thesis. This subsection basically represents a summary of the whole approach, showing the workflow by means of which a Service Graph is optimally enriched with network security defences from cybersecurity attacks.

First of all, Figure 8.2 represents the Service Graph that has been defined by means of a composition of only service network functions, without any specific constraints regarding the automatic allocation of the firewalls. This network service is then translated into the Allocation Graph showed in Figure 8.3; since between any pair of nodes an Allocation Place is created, in this example the complete solution space will be explored to achieve the optimal allocation schema.

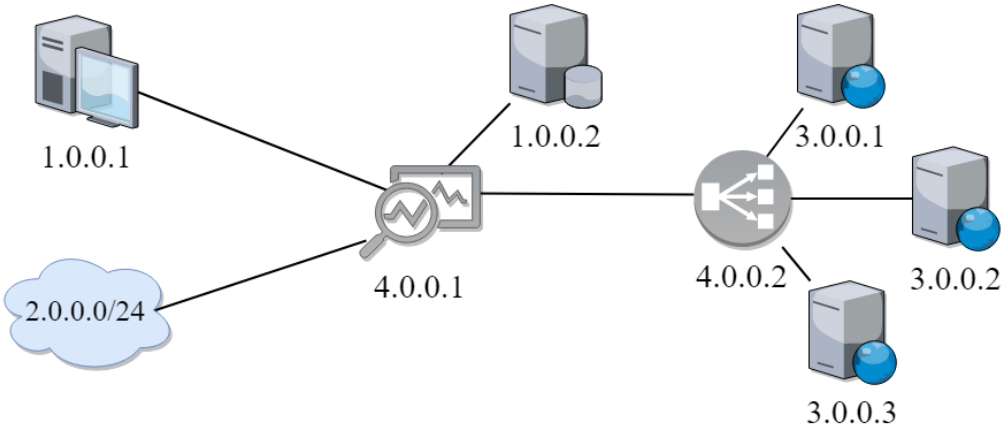


Figure 8.2. Service Graph of the clarification example

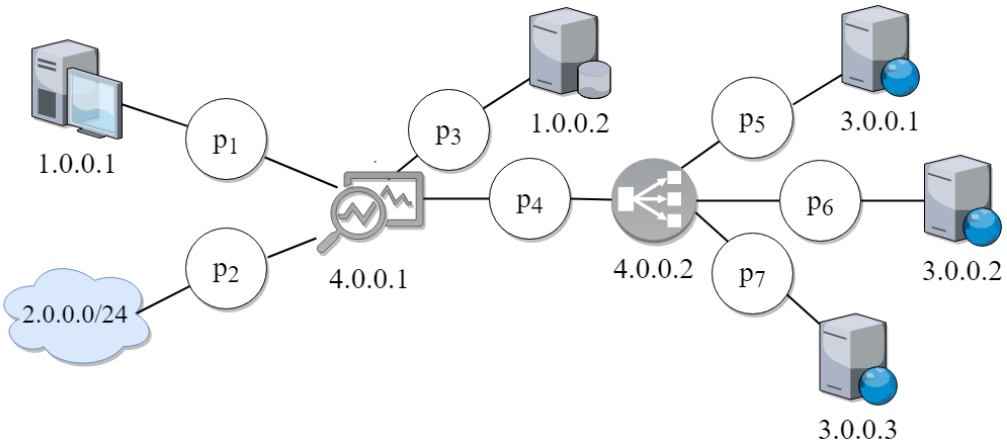


Figure 8.3. Allocation Graph of the clarification example

Secondly, Figure 8.4 represents a set of Network Security Requirements that have been defined alongside a *specific* general behaviour and that represent which traffic flows must be denied or allowed inside the service.

According to the objectives explained in this section, finally the outcome which is provided is the result showed in Figure 8.5; in fact, a correct solution could be achieved and a non-enforceability report was not needed.

Type	IPSrc	IPDst	portSrc	portDst	transportProto
Isolation	1.0.0.1	3.0.0.*	*	*	*
Isolation	1.0.0.1	2.0.0.*	*	*	*
Isolation	2.0.0.*	3.0.0.*	*	*	*
Isolation	1.0.0.2	3.0.0.*	*	*	UDP
Isolation	1.0.0.2	3.0.0.*	*	90	TCP
Reachability	1.0.0.2	3.0.0.*	*	80	TCP
Reachability	2.0.0.*	1.0.0.1	*	*	*

Figure 8.4. Network Security Requirements of the clarification example

From the outcome, it is possible to underline that in the optimal allocation schema two firewalls are needed, because the Service Graph has a ramified architecture where the traffic flows pass through different paths, differently from a traditional Service Function Chain. Nevertheless, if this task had been performed manually, it could have led to allocate more than two firewalls, i.e. one firewall before each end point of the service. The final solution could have been correct, but not optimal, leading to resource consumption for the servers where the VNFs would have been deployed.

Then, with regards to the auto-configuration of the firewall policies, each one is characterized by a single specific rule, in addition to a *DENY* default action that is characteristic of a *whitelisting* mode. In particular, fw_2 allows all the TCP traffic targeted to the destination port with number 80, blocking all the other communications. A manual creation of rules which must consider several combinations of the IP quintuple components is not a trivial task, particularly when the number of rules increases. Moreover, in a distributed architecture where multiple firewall instances are deployed, it becomes difficult not to make human errors while configuring the

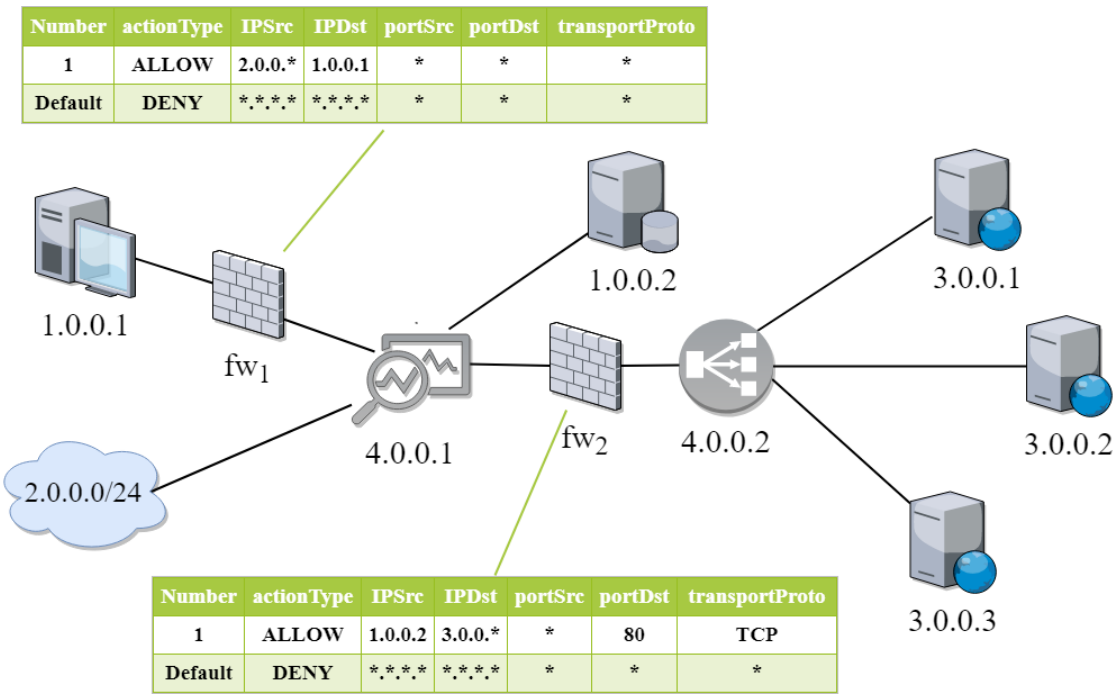


Figure 8.5. Expected outcome of the clarification example

rules in more than a single firewall.

Moreover, it is possible to underline that the *wildcards* feature is often used by the solver in the definition of the policy rules, proving itself as a fundamental element to improve the expressiveness of the framework and to minimize the total number of rules that must be configured.

Chapter 9

Results

In this chapter the results of the performance tests carried out on the developed framework are illustrated, in order to show which goals have been achieved and to understand which limitations should be refined in the future.

The structure of this chapter is the following:

- in Section 9.1 useful terminology to understand the scenarios in which the framework has been tested is provided;
- in Section 9.2 a comparison with the previous implementation of the framework is showed;
- in Section 9.3 the comparison in terms of performance between different working conditions is showed, such as a comparison between chain and graph, between blacklisting and whitelisting working modes or between isolation and reachability requirements;
- in Section 9.4 the impact of Allocation Nodes which are not possible placeholders, as illustrated in Section 9.1, will be investigated;
- in Section 9.5 the results of scalability tests are illustrated to understand which are the most relevant problems to be addressed in the future.

All the tests have been performed on a workstation with an Intel Core i5 CPU at 2.40-3.00 GHz and 4.00 GB of RAM, with the exception of the scalability tests of Section 9.5, for which a more powerful machine with an Intel Core i7 CPU at 3.40 GHz and 32.00 GB of RAM has been exploited.

9.1 Useful terminology

The implementation of the Allocation Graph concept in the framework, as it has been described in Section 6.2, introduced the possibility to perform an optimal allocation of Network Security Functions in a multiplicity of nodes inside a logical topology automatically generated from a Service Graph; the independence of this

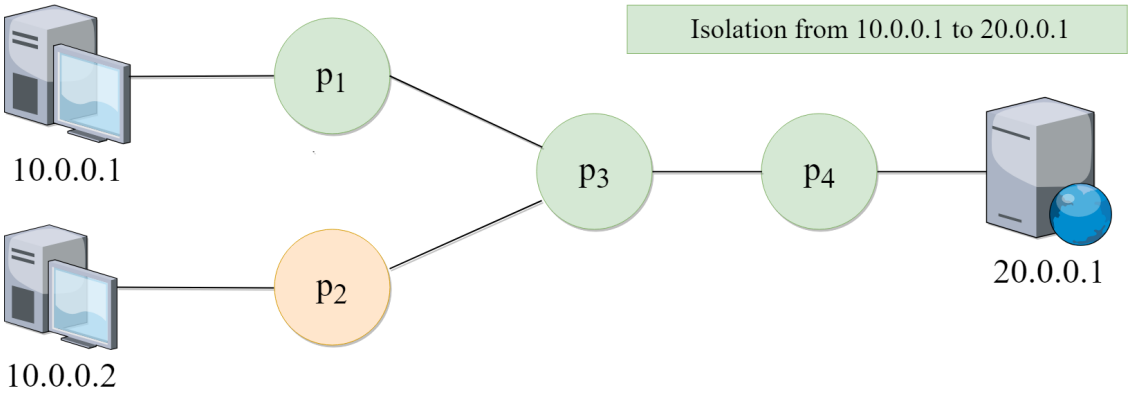


Figure 9.1. Allocation Graph example to explain useful terminology.

abstract network from the substrate hardware infrastructure represents a key aspect of the *SDN* and *NFV* principles.

With the goal to provide a better comprehension of how the developed features were tested, Figure 9.1 presents an Allocation Graph example which will be now used to present useful terminology for understanding the results of performance tests. In this example only a Network Security Requirement is established, that is an isolation requirement between the web client with IP address 10.0.0.1 as a source and the web server responding at the IP address 20.0.0.1 as the destination. Despite the presence of another web client with address 10.0.0.2, the specified Network Security Requirement is not influenced by packets sent by this end point, which are not considered by the *ADP* module in solving the MaxSAT problem for the optimal allocation of firewalls and the automatic configuration of Filtering Policy rules.

In this example, all the nodes of the Allocation Graph - both *green* and *orange* - are called ***Allocation Nodes***: they represent the complete topology which VEREFOO must analyse before solving the optimization problem, independently from the characteristics of the specified Network Security Requirements.

Nevertheless, with a careful look at this scenario, it becomes evident how the *orange* Allocation Node will not be considered by the *z3Opt* engine as a possible choice where an instance of a packet filter would be allocated; in fact, the packets which the client 10.0.0.1 sends towards the corresponding web server 20.0.0.1 will not ever be able to reach it by passing through the *orange* node. For comparison, if in another scenario an additional Network Security Requirement involving the client 10.0.0.2 is introduced, then also this node is a potential candidate for the allocation phase.

Consequently, the term ***Allocation Places*** or ***Placeholders*** is effectively used only to represent the subset of Allocation Nodes - represented with the *green* colour in the provided example - which, inside the complete logical topology, the *ADP* module will effectively consider as candidates for the allocation of firewalls, after

a preliminary visit of the graph based on the source and destination specifications of all the Network Security Requirements. Normally, if the service designer does not make any significant mistake in defining the Service Graph and the Network Security Requirements, in this typical scenario all the Allocation Nodes are actually Allocation Places and there is no distinction between the two terms.

For this reason, with the only exception of the results showed in Section 9.4, all the other tests have been carried out in scenarios where all the Allocation Nodes are also possible Allocation Places for Network Security Functions. This choice is motivated, as it will be further explained in the dedicated section, by the fact that in future the impact due to the present of Allocation Nodes not involved in any Network Security Requirement could be severely minimized, since it is exclusively related to the presence of a set of hard constraints of the MaxSMT problem.

9.2 Comparison with old framework

The old version of the framework was tested with the maximum numbers of 3 packet filters – fixed on specific positions – and 4 Network Security Requirements, because increasing these numbers would lead to a not acceptable, long time before the ending of the run; the main performance limitations were, in fact, due to a not appropriate usage of the *wildcards* feature as illustrated in Section 7.4 and scalability issues, since the specification of soft constraints was not minimized and pruning strategies were not adopted. Moreover, the formal model of the auto-configuration of firewalls

Figure 9.2. Results of performance tests of the old model

was characterized by redundant soft clauses and the absence of the Allocation Graph did not allow the framework to automatically decide which are the best positions for the packet filters.

The only scenario considered for this test is the scenario represented in Figure 5.5 of Section 5.4.1 because the old framework did not support the existence of the Allocation Graph; consequently, the provided input is the Service Graph where the packet filters are installed on specific nodes and the *ADP* module must provide only the auto-configuration of their Filtering Policies, without exploiting the allocation feature. This evidently represents a limitation for the new version of the framework, because the clauses of the MaxSMT problem have been designed assigning the automatic allocation a central role, by means of the high weight assigned to the soft constraint represented by Formula 8.1.

The comparison was performed in a scenario where a group of clients are linked to a server by means of a chain of packet filters; for each pair client-server an isolation requirement is specified and the working mode of the firewalls is blacklisting.

Figure 9.2 shows that the old version of the framework had severe scalability issues when both the number of packet filters and the number of Network Security Requirements increase. The first issue is related to the fact that redundant soft constraints were modelled for the auto-configuration of a rule (e.g. instead of a single soft constraint to represent the absence of a rule, eight clauses were originally present in the MaxSMT problem), while the second problem is related to the implementation of forwarding rules with quantifiers and the absence of pruning

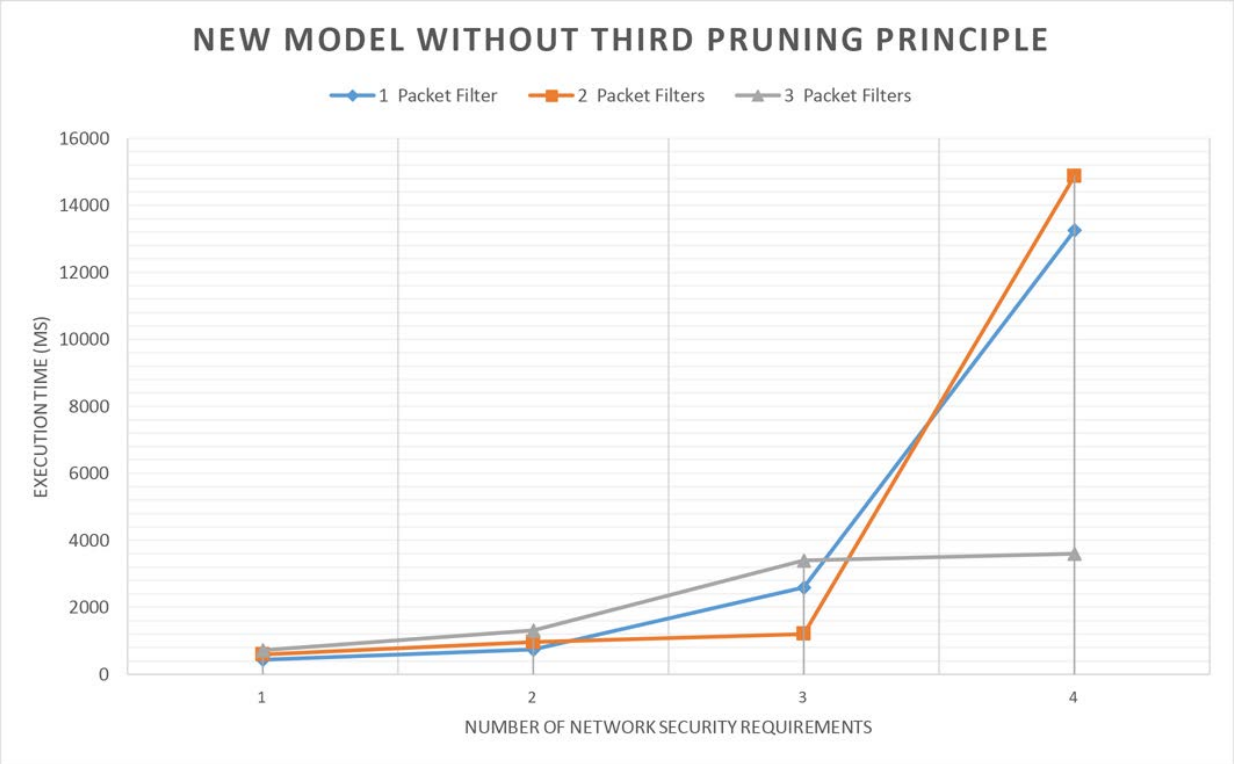


Figure 9.3. Results of performance tests of the new model without the third pruning principle

algorithms to reduce the number of rules which could be configured on a specific packet filter according common sense principles.
Furthermore, if this old version is tested with a bigger number of packet filters or Network Security Requirements, the optimization problem becomes unfeasible, clearly showing the limitations of this implementation.

Figure 9.3, instead, shows the results achieved with the *ADP* module, used exclusively in distribution phase as mentioned, without the algorithm based on the third pruning principle described in Subsection 8.6.1 and that would exploit the *wildcards* feature to further improve the performance; in this way, only the improvements in terms of design of the constraints can be evaluated.
For the most complex example (3 packet filters and 4 Network Security Requirements), the time is much inferior than the results got with the previous implementation; even though in each packer filter all the four policy rules are tentatively placed, *z3Opt* is able to efficiently parse all the relaxable and not-relaxable clauses. If in scenarios designed for scalability tests the absence of pruning and heuristics can be critical, for simple networks the *ADP* already shows better performance without them.

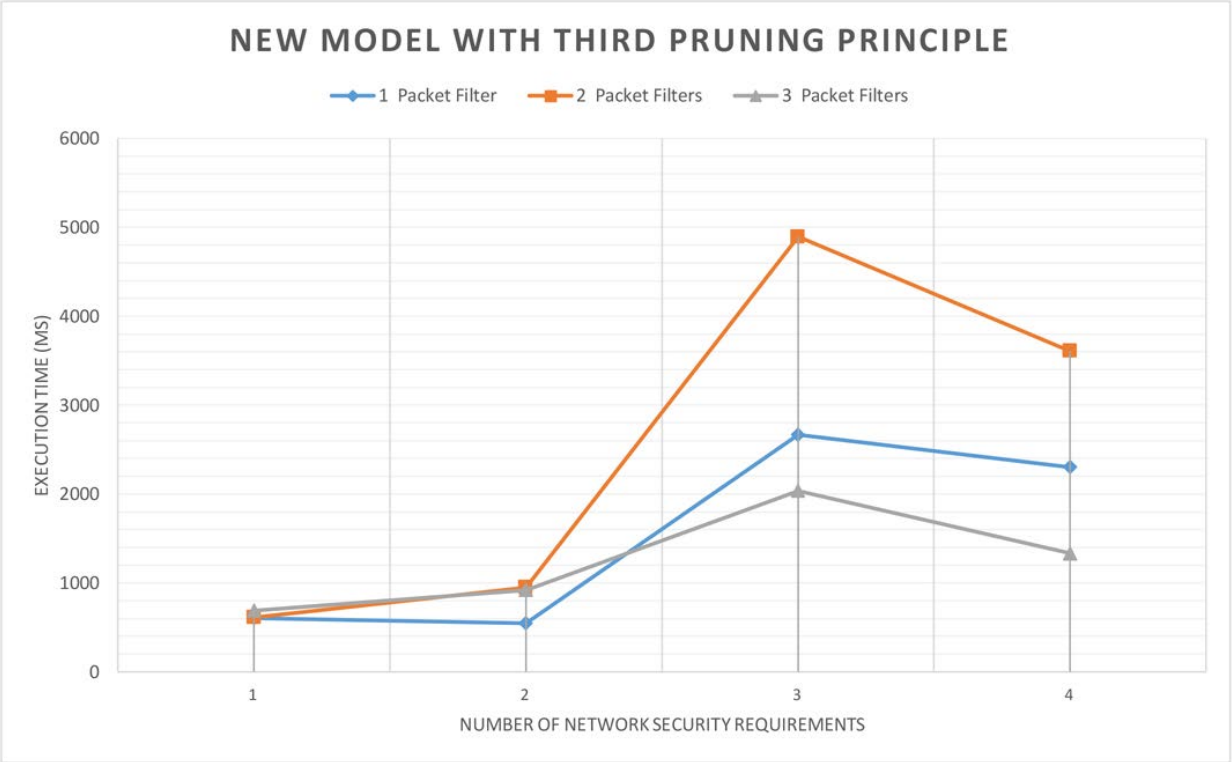


Figure 9.4. Results of performance tests of the new model with the third pruning principle

Finally, Figure 9.4 shows the results achieved when the third principle of firewall auto-configuration algorithm was enabled. In this case all the execution time estimations are not superior than 5 seconds, because the usage of wildcards - which requires more soft constraints - is compensated by the minimization of the rules which can be configured on any packet filter. It is besides interesting to notice how the required time for the pre-processing computation is always negligible compared

to the time spent by the optimizer engine; the same consideration will apply also to all the next performance tests.

It is worth mentioning that, as it will be shown in Section 9.5, the new framework implementation can achieve more scalable results also in scenarios with a number of packet filters and Network Security Requirements bigger than, respectively, three and four, since the whole design phase considered scalability as a central aspect; in those scenarios, the old implementation did not properly work and an extended comparison cannot be carried out – in addition, also the allocation phase introduced in the new implementation cannot be compared with the most similar feature of the old framework, which was the removal of useless firewall, because they are based on a different approach and the second one has again scalability issue.

9.3 Comparison between different working conditions

In this section, all the performance tests have been carried while the reduction algorithm based on the *enforce* function, described in Subsection 8.6.1, is disabled in order to create the worst conditions. In fact, if active, it would prune some decisions of the locations where some rules could be configured - this action has a clear benefit for the effective usage of the framework, but it would simplify the results of the performance tests without making it possible to understand the impact of the constraints of the z3 model.

The most common working conditions which have been studied are the following:

- in Subsection 9.3.1 a comparison between two different kinds of logical topology, a ramified graph and a linear chain;
- in Subsection 9.3.2 a comparison between the possible working modes of a packet filter, whitelisting and blacklisting;
- in Subsection 9.3.3 a comparison between the enforcement of isolation and reachability requirements.

All the comparisons have been evaluated by progressively increasing both the number of Allocation Places and of Network Security Requirements from a scenario to the next one, even though the increase of the placeholders is higher than the corresponding increase of the Network Security Requirements because of difference scalability, as it will be showed in Section 9.5 in more details.

9.3.1 Comparison between graph and chain

The chain is a linear topology where the total number of Allocation Nodes coincide with the number of possible Allocation Places and where for each Network Security Requirement the packets generated in the model for the functions allocation and for the configuration of the Filtering Policies cross every node. On the other hand,

a graph is characterized by a ramified structure, where clients and servers are attached to different nodes and the paths are not necessarily overlapping. These two topologies have been tested with whitelisting packet filters and isolation requirements.

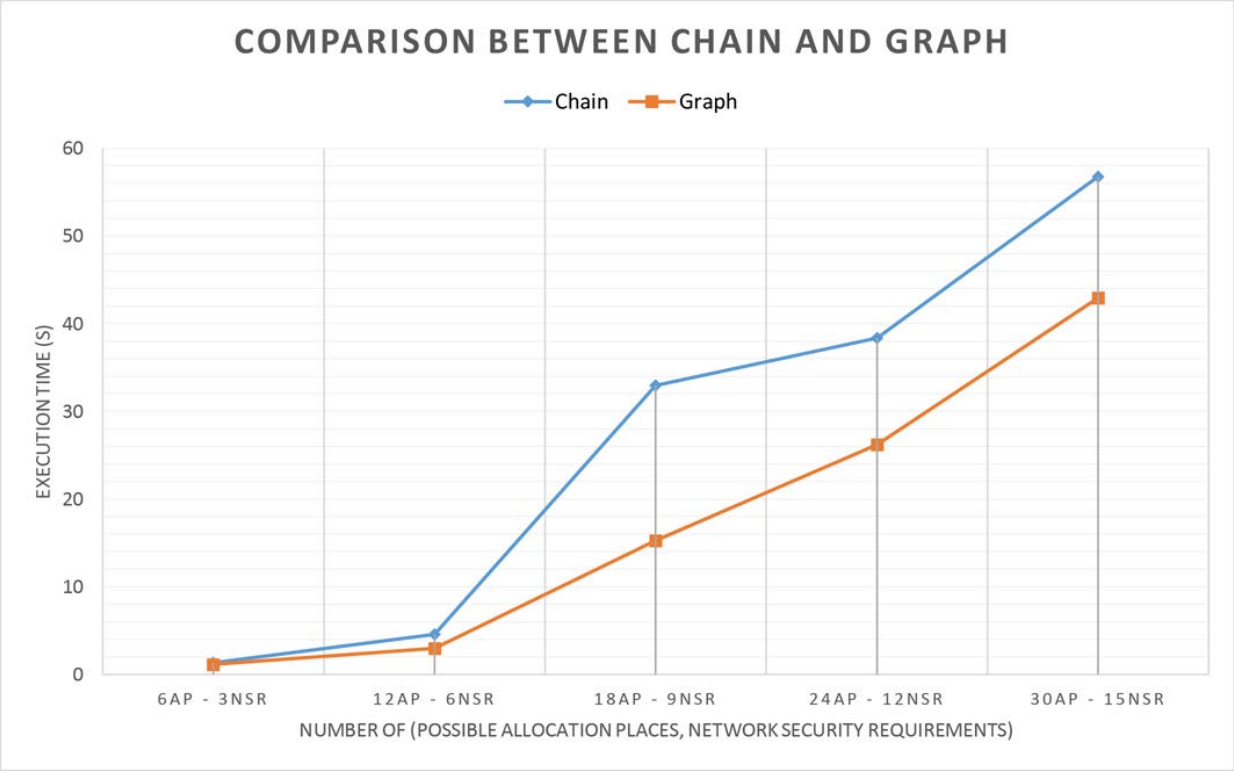


Figure 9.5. Results of performance tests between chain and graph

The results in Figure 9.5 shows as performance is better when the enforcement of the Network Security Requirements is performed in a ramified graph; this difference is not detectable in very small scenarios, but when the number of Allocation Places is at least 18 and the number of Network Security Requirements is around 9 the computation time to auto-configure and allocate packet filters in a chain is around the double than in a graph.

The reason of these results is that, in a chain, all the nodes are crossed by packets related to each Network Security Requirement, while in a ramified graph each packet passes through a limited set of Allocation Places. Consequently, the number of hard constraints is different; we find this way an experimental proof that not only the soft, but also the hard clauses are critical for the performance of the framework implementation. In the scenario of a chain, the number of hard constraints per node is higher but they do not contribute in a reduction to the solution space, because instead they define additional events of receiving and sending packets.

Consequently, all the future comparisons will be carried out in Allocation Graphs modelled as chains, in order to further understand the capabilities of the framework in a worse condition, which nonetheless is easily subject to perform scalability tests.

9.3.2 Comparison between whitelisting and blacklisting

The service designer, when using the framework, can decide the default action, i.e. the working mode, of the packet filters the *ADP* module will tentatively place in the Allocation Graph; the choice is among whitelisting, if the firewall drops all the packets without further specifications about, and blacklisting, if the firewall lets the packets pass through unless exceptions are defined. It is also possible to specify different default actions for each packet filter instance, even though a similar configuration could easily lead to conflicts and makes the result more difficult to understand or debug. If, on the other hand, the service designer does not provide any indication as input, the framework adopts whitelisting as preferred mode, since it is the working mode which allows a higher level of security because it is not necessary to strictly identify all the kinds of traffic flow which must be prohibited.

Inside the z3 model, the formulas which have been built for the two different working modes are very similar and they correspond to the same number of constraints, as it has been showed in Subsection 8.6.2. As a consequence, the expected performance should be similar. This prevision has been proved in a topology where isolation requirements must be enforced and where all the Allocation Places are potentially crossed by a packet relative to the requirements, unless a firewall instance is installed to block it.

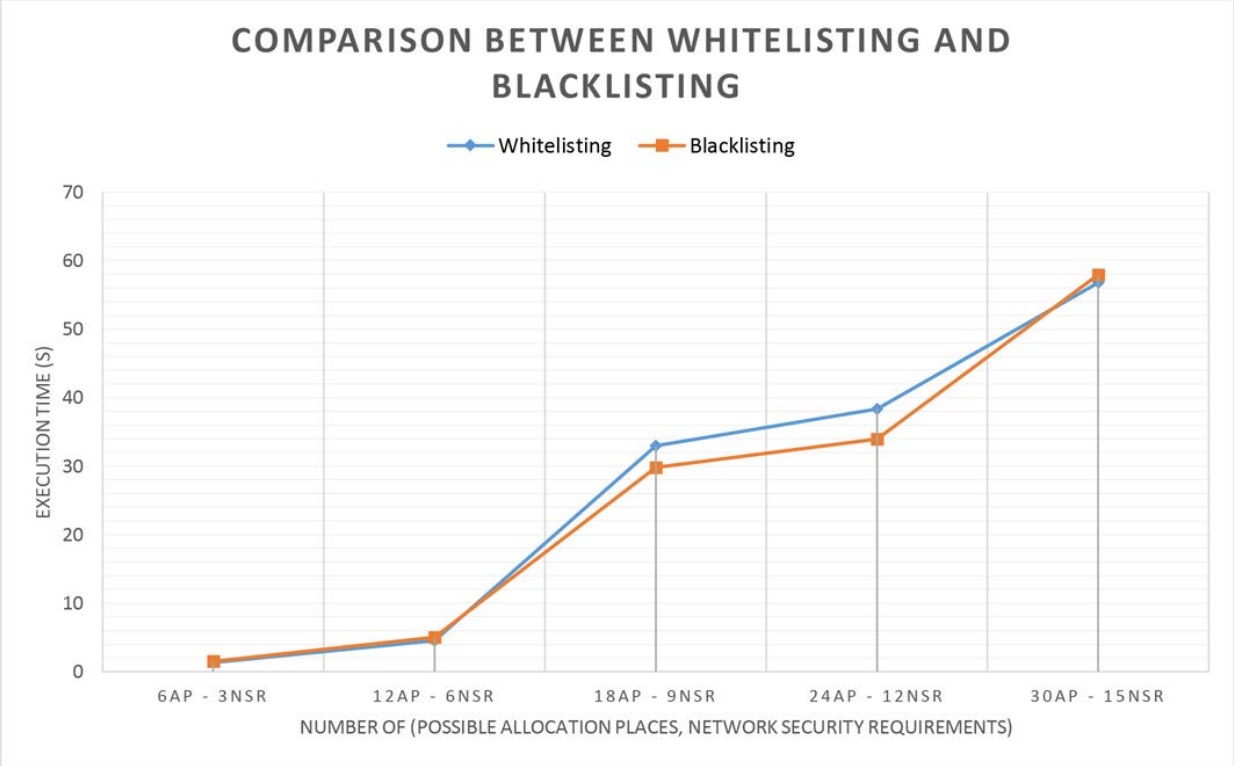


Figure 9.6. Results of performance tests between whitelisting and blacklisting

The results are showed in Figure 9.7. Even when increasing the number of Allocation Places and of Network Security Requirements, the differences in terms of performance are negligible. This confirms the hypothesis that the choice between

these two different working modes does not have a central impact on the execution time of the framework, but the only consequence is the different issues each one brings with it, i.e. whitelisting is more difficult to configure accurately, while blacklisting offers a lower security level if the requirements are not fully correctly specified.

9.3.3 Comparison between isolation and reachability

When the service designer decides which kinds of Network Security Requirements to specify as input for VEREFOO, he can choose between the isolation requirement, if he wants to avoid a specific communication, or reachability requirement, if he wants to allow the relative kind of traffic. As it has been described in Sections 7.5 and 7.6, these two types of Network Security Requirements are modelled in the z3 instance of the MaxSMT problem with different kinds of hard constraints, so an interesting test case which has been studied is relative to the comparison between a scenario where only reachability requirements are specified with another scenario where, instead, the service designer only introduced some isolation requirements.

The tests relative to this comparison have been carried out on an Allocation Graph, where the number of Allocation Places coincides with the total number of Allocation Nodes, the packets generated by each Network Security Requirement cross every Allocation Place and all the firewalls which can be allocated are configured in whitelisting mode.

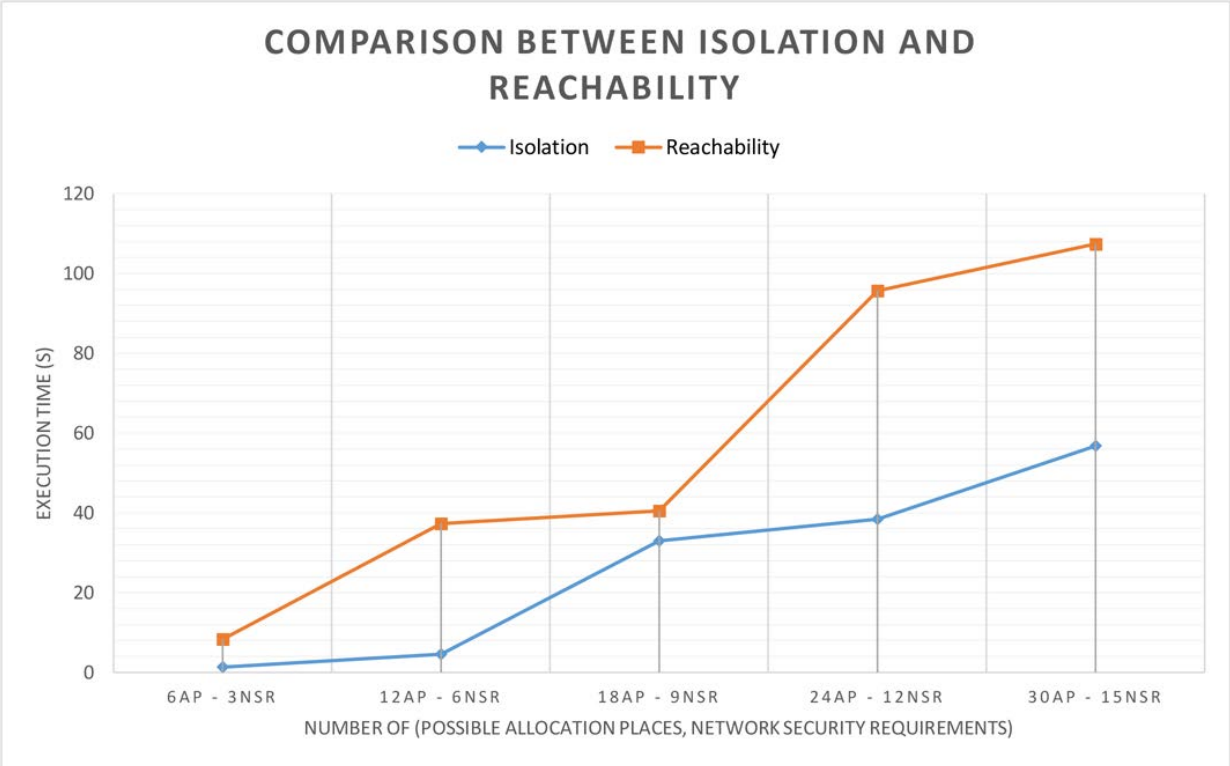


Figure 9.7. Results of performance tests between isolation and reachability.

The results of the tests are showed in Figure 9.7. Since the simplest Allocation

Graphs where the number of Allocation Places and Network Security Requirements are limited, it is evident how the computation time required by the framework to compute a correct allocation schema and configuration of the needed firewalls is generally bigger for reachability requirements than for isolation requirements in the tested conditions.

This experimental result can be explained by the fact that, in case a reachability must be enforced, the related packets must effectively cross all the allocated firewalls, while in the case of an isolation requirement the packet could be blocked by one of the first firewalls it encounters. Furthermore, when the hard constraints are specified for the reachability, it is requested that the destination receives a packet whose characteristics respect the values of the specified requirement. This is a pressing constraint because it forces the existence of this packet to be received; on the other hand, in the isolation case, the corresponding constraint is that no packet received by the destination should match the input requirement, without requiring the specific existence of a received packet.

9.4 Evaluation of Allocation Nodes number impact

In all the previous tests, the number of Allocation Nodes, which according to the definitions provided in Section 9.1 were not possible Allocation Places or Placeholders, was equal to zero, because every node was interested by at least a Network Security Requirement and crossed by at least a packet for the configuration of the Filtering Policy rules. This section focuses, instead, on scenarios where the logical topology the service designer provides as input is characterized by nodes which will not be considered by the *ADP* module in the allocation and configuration tasks.

Theoretically, the number of these Allocation Nodes should not have any impact on the performance of the framework because the construction of the model for the optimizer engine does not include either any soft constraint about the allocation of firewalls and configuration of policy rules nor any hard constraint for expressing the forwarding rules, since their routing tables are basically empty after the recursive visit of the graph. However, the experimental results do not match with this prediction, as it is shown by Figure 9.8 and Figure 9.9, where respectively the impact of these Allocation Nodes is evaluated considering as fixed parameter the number of Network Security Requirements or possible Allocation Places.

The two charts show how, if the number of total Allocation Places is 50, the resulting performance is similar to the scenarios with the same working conditions where the Allocation Nodes which are not placeholders do not exist; the overhead which they introduce is in fact negligible. On the other hand, doubling their number to 100, the execution time of the framework doubles as well, also for the simplest topology with a limited number of placeholders and Network Security Requirements.

These results can be explained by the fact the, despite for the optimization problem these nodes do not have any impact, however for each one of them a limited set

Figure 9.8. Results of performance tests for the evaluation of Allocation Nodes impact, with fixed Network Security Requirements number

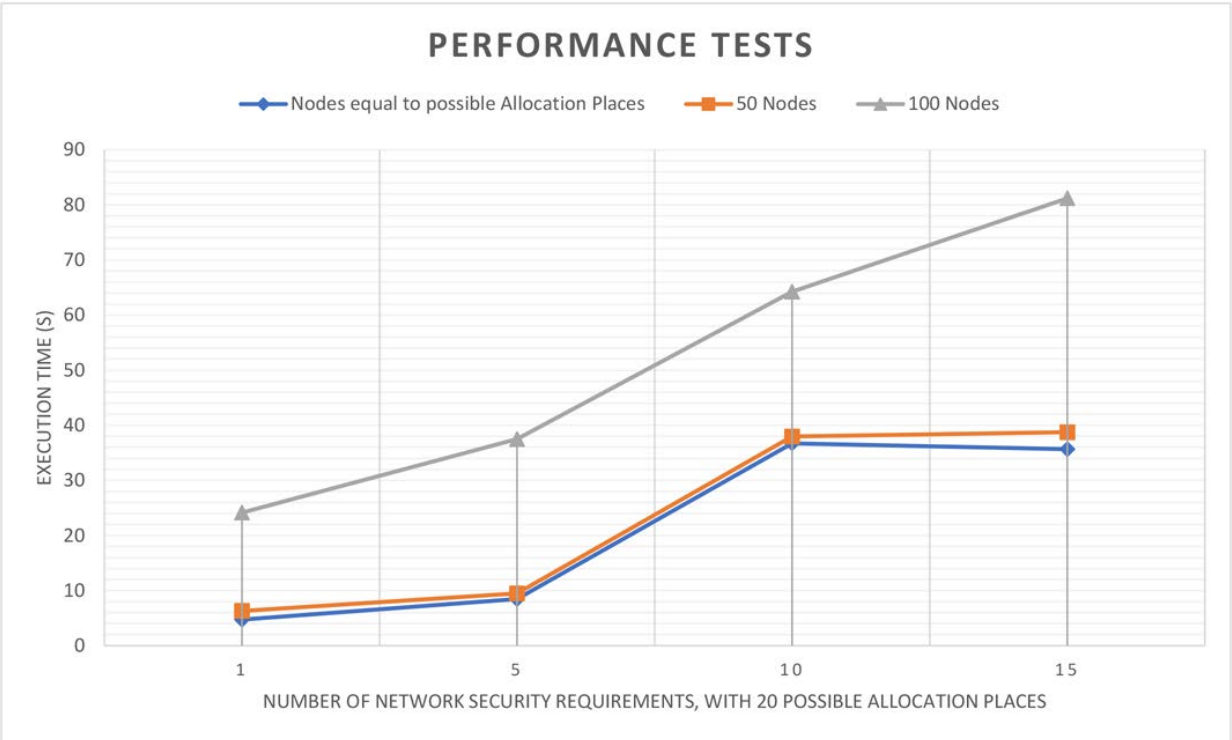


Figure 9.9. Results of performance tests for the evaluation of Allocation Nodes impact, with fixed Allocation Places number

of hard not-relaxable constraints are built; some examples are the constraint which correlates the unique identifier of the node with the z3 data structure representing its IP address and the constraints which in the definition of the packet data type structure make use of quantifiers as node variables.

If in the current state of the work it could represent a potential issue, not only it is related to very big scenarios where the service designer introduces a big number of useless Allocation Nodes, but since the overhead is limited to the hard constraints a further refinement of the inner implementation of the framework could lead to the elimination of the aforementioned hard constraints, since they do not have any utility.

9.5 Scalability tests

Finally, a series of scalability tests have been carried out in scenarios where the number of Allocation Places is the same as the number of Allocation Nodes, the requested Network Security Requirements are isolation properties and all the firewalls which can be allocated to satisfy them are configured in whitelisting mode.

The three metrics which have been identified as the most critical for the scalability tests are the following:

- number of Allocation Places where the firewalls can be allocated;
- number of input Network Security Requirements;
- number of assertions – including both soft and hard constraints – in the z3 instance of the MaxSMT problem.

First of all, focusing on the first two metrics – numbers of Allocation Places and of Network Security Requirements –, the approach which has been followed to perform the scalability tests has been to increase one metric to a higher value, while the other is kept 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 9.10 for the Allocation Places, whereas in Figure 9.11 for the Network Security Requirements.

First of all, by analyzing the two charts, it is evident that the increase of the execution time is not exponential, independently from which metric is considered (Allocation Places or Network Security Requirements); this is fundamental result, given the intrinsic computational cost of a MaxSMT problem. Moreover, the framework is able to manage also Allocation Graphs of bigger sizes, where more end points or service functions are present, and the MaxSMT solver can achieve the optimal allocation schema and configuration of the firewalls, if the number of Network Security Requirements is not really high.

It is, then, possible to notice that an increment of the Network Security Requirements number produces a higher computation time then the same increment

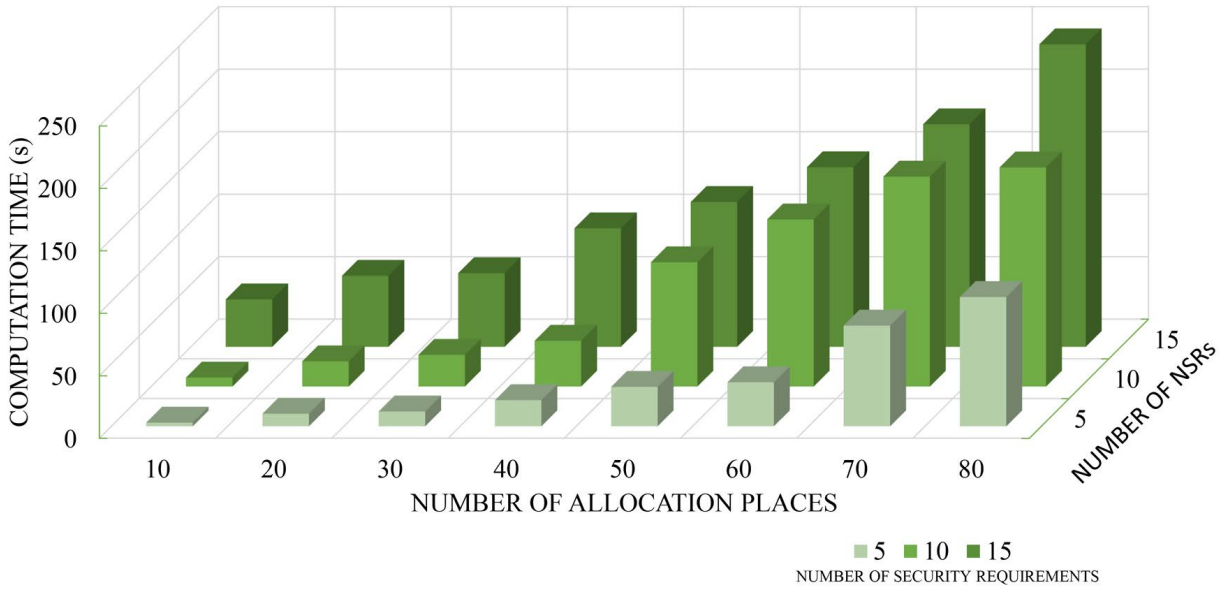


Figure 9.10. Results of scalability tests for Allocation Places

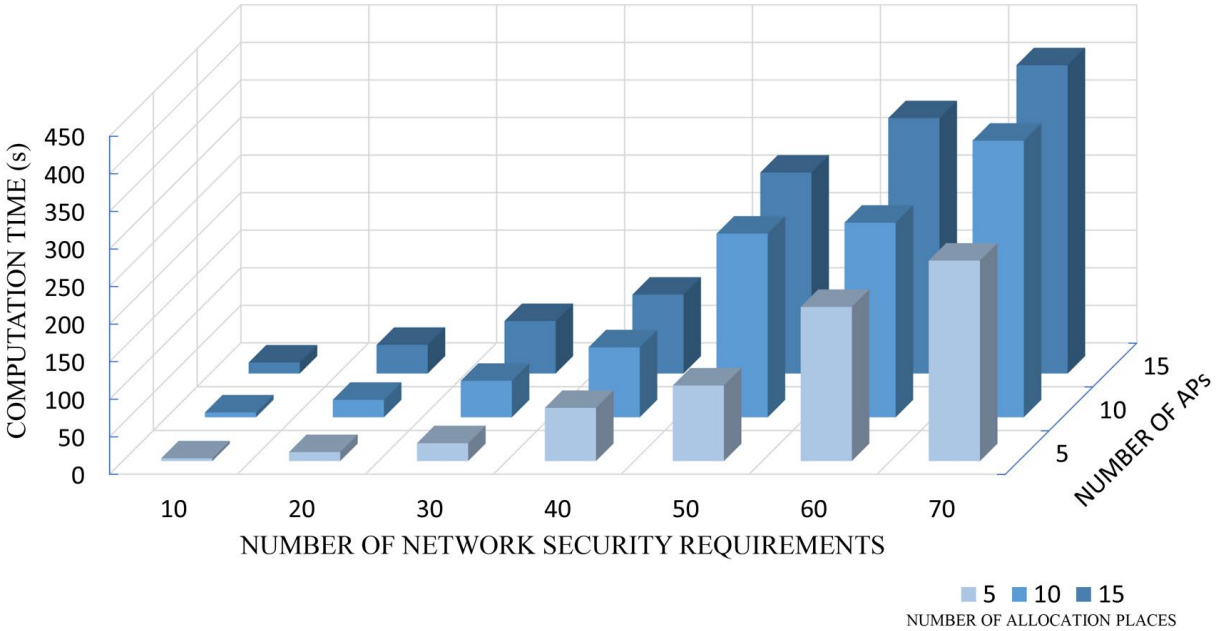


Figure 9.11. Results of scalability tests for Network Security Requirements

of the Allocation Places number. The reason of this experimental result is that the verification of the satisfiability of a NSR must be carried out in all the service by the creation of a packet which flows across all the possible paths, whereas the constraints introduced by an Allocation Place and the corresponding firewalls only concern the traffic flow which crosses it.

Finally, the third metric – the number of assertions of the MaxSMT problem – has been considered, exploiting the results achieved with the scalability tests carried out for the other two metrics to build the chart showed in Figure 9.12, retrieving for

each instance the number of assertions of the corresponding z3 model. This metric, actually, is related to the previous two, since the number of soft constraints depend on how many Allocation Places are present in the service, whereas the number of hard constraints depends on the cardinality of the Network Security Requirements set.

Figure 9.12. Scalability tests on assertions

This chart shows that the increase of the computation time is typically linear also when the cardinality of the MaxSMT clauses set increases; the result is achieved considering as fixed both the input requirements, so that bigger number of clauses is exclusively caused by constraints that are related to the service composition or to the firewalls that have been allocated. It is important to underline, actually, that this number includes both the soft and hard constraints of the optimization problem, because not all the hard constraints allow to limit the size of the solution space, but some of them increase this size and can have an impact on the performance which is the comparable to the impact of a soft constraint – which, evidently, requires additional computation time because the optimizer must try to satisfy all the soft clauses to reach the optimal solution.

Chapter 10

Conclusions

During the thesis work, major improvements of the *ADP* module of VEREFOO have been designed and developed, with the purpose of extending the capabilities of the framework and, at the same time, achieving better performance results than the previous version. This work has been performed with the future goal of using the developed framework in an NFV environment, by exploiting all the advantages provided by the decoupling between the virtual functions and the physical infrastructure.

Firstly, a complete study about which are the most important scenarios of usage for the framework has been carried out, to understand which components were missing and which other elements required be improved to be compliant with the proposed goals. After identifying them, the focus has been on the design and development of the allocation and configuration phases for the firewalls, which represent one of the most important Network Security Functions.

Then, a new internal level of abstraction has been introduced through the design and the implementation of the Allocation Graph, a logical topology coming from the Service Graph where in some placeholders the firewalls can be allocated to satisfy the input Network Security Requirements; furthermore, an automatic mechanism for the creation of the Allocation Graph has been defined. Then, the forwarding rules by means of which the firewalls can forward the received packets have been completely remodelled, so that the usage of the *quantifiers* feature for the nodes of the graph has been removed and better scalability has been reached.

Moreover, after defining a more complete model of the connectivity Network Security Requirements which a service designer can specify as input to VEREFOO, the model of the isolation property has been changed because the previous one was able to properly support exclusively chains, while the model of the reachability property has been refined to avoid the use of *quantifiers* and to be compliant with the introduction of the Allocation Graph and with the richer model of the security requirements themselves. While implementing these novelties, further extensions have been introduced, such as the possibility to specify a server as a source of a Network Security Requirement and a client as a destination, allowing bidirectional communications and the creation of richer scenarios. In addition, the *wildcards* feature has been modified so that it is now effectively usable in the framework.

A central work has subsequently been to define the formal model of the automatic allocation of firewalls on the placeholders of the Allocation Graph and to change the model of the automatic configuration, since the precedent one led to not good performance when exploiting the wildcards. In this context, some pruning strategies have been pursued and introduced, to reduce the number of soft and hard constraints which must be defined in the z3 model for the configuration of firewalls, given the input set of Network Security Requirements.

After all these works have been concluded, a series of performance tests have been carried out both to understand the differences in terms of computation time between the possible working conditions and to study the scalability achieved by the new version of the framework. About this aspect, the result which has been achieved is that the scalability is quite improved in respect with the previous implementation, both about the placeholders' number where the firewalls can be allocated and the input Network Security Requirements number.

Future works about the *ADP* module of VEREFOO could be to introduce a larger set of network functions which the service designer can exploit to define the Service Graph, to model additional Network Security Functions such as anti-spam filters and Web Application Firewalls to enrich the capabilities of the framework and to define new types of Network Security Requirements which could consider additional features – e.g. at application layer – of the communications to allow or to block.

Bibliography

[1] C. Basile, A. Lioy, C. Pitscheider, F. Valenza, and M. Vallini, "A novel approach for integrating security policy enforcement with dynamic network virtualization," in *Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015, London, United Kingdom, April 13-17, 2015*, 2015, pp. 1–5. [Online]. Available: https://doi.org/10.1109/NETSOFT.2015.7116152

[2] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool, "*Firmato*: A novel firewall management toolkit," *ACM Trans. Comput. Syst.*, vol. 22, no. 4, pp. 381–420, 2004. [Online]. Available: https://doi.org/10.1145/1035582.1035583

[3] J. M. Halpern and C. Pignataro, "Service function chaining (SFC) architecture," *RFC*, vol. 7665, pp. 1–32, 2015. [Online]. Available: https://doi.org/10.17487/RFC7665

[4] D. Kreutz, F. M. V. Ramos, P. J. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015. [Online]. Available: https://doi.org/10.1109/JPROC.2014.2371999

[5] R. Chayapathi, S. F. Hassan, and P. Shah, *Network Functions Virtualization (NFV) with a Touch of SDN*. Addison-Wesley Professional, 2016.

[6] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for policy-based management," *RFC*, vol. 3198, pp. 1–21, 2001. [Online]. Available: https://doi.org/10.17487/RFC3198

[7] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model - version 1 specification," *RFC*, vol. 3060, pp. 1–100, 2001. [Online]. Available: https://doi.org/10.17487/RFC3060

[8] A. Matheus, "How to declare access control policies for XML structured information objects using oasis' extensible access control markup language (XACML)," in *38th Hawaii International Conference on System Sciences (HICSS-38 2005), CD-ROM / Abstracts Proceedings, 3-6 January 2005, Big Island, HI, USA*, 2005. [Online]. Available: https://doi.org/10.1109/HICSS.2005.300

[9] F. Valenza and A. Lioy, "User-oriented network security policy specification," *J. Internet Serv. Inf. Secur.*, vol. 8, no. 2, pp. 33–47, 2018. [Online]. Available: https://doi.org/10.22667/JISIS.2018.05.31.033

[10] J. D. Moffett and M. S. Sloman, "Policy hierarchies for distributed systems management," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1404–1414, Dec 1993.

[11] J. Zhou and J. Alves-Foss, "Security policy refinement and enforcement for the design of multi-level secure systems," *Journal of Computer Security*, vol. 16, no. 2, pp. 107–131, 2008. [Online]. Available: http://content.iospress.com/articles/journal-of-computer-security/jcs300

[12] A. K. Bandara, E. Lupu, J. D. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), 7-9 June 2004, Yorktown Heights, NY, USA*, 2004, pp. 229–239. [Online]. Available: https://doi.org/10.1109/POLICY.2004.1309175

[13] N. B. Youssef and A. 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*, 2011, pp. 461–466. [Online]. Available: https://doi.org/10.1109/CIT.2011.84

[14] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[15] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. 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*, 2017, pp. 359–373. [Online]. Available: https://doi.org/10.1145/3132747.3132753

[16] K. Adi, L. Hamza, and L. Pene, "Automatic security policy enforcement in computer systems," *Computers & Security*, vol. 73, pp. 156–171, 2018. [Online]. Available: https://doi.org/10.1016/j.cose.2017.10.012

[17] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[18] "Verigraph repository," https://github.com/netgroup-polito/verigraph, accessed: 2019-04-29.

[19] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, "Formal verification of virtual network function graphs in an sp-devops context," in *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, 2015, pp. 253–262. [Online]. Available: https://doi.org/10.1007/978-3-319-24072-5_18

[20] B. E. Carpenter and S. W. Brim, "Middleboxes: Taxonomy and issues," *RFC*, vol. 3234, pp. 1–27, 2002. [Online]. Available: https://doi.org/10.17487/RFC3234

[21] "Z3 programming guide," http://theory.stanford.edu/~nikolaj/programmingz3.html, accessed: 2019-04-29.

Appendices

Appendix A

z3 Java API Manual

In this Appendix some guidelines about how to use the Java z3 APIs are provided in order to help the readers and future maintainers of the framework to have a better idea of how a MaxSMT problem can be formulated using the z3 optimizer engine. A complete guide about how to use the z3 language, independently of the high-level programming APIs adopted, can be found in [21].

A.1 Context class

The **Context** class represents the core of every z3 model, independently of the usage of the optimizer engine; it allows the definition of new types, internally called *sorts*, and new expressions of existing or user-defined data types.

A.1.1 Creation of data types (*sorts*)

To retrieve a built-in traditional data type, so that it can further be used for the creation of new expressions, methods which should be invoked are named *mkType-Sort*, where *Type* is the name of the specific data type. Relevant examples are the following:

- *mkBoolSort* to represent a boolean type;
- *mkIntSort* to represent an integer number type;
- *mkRealSort* to represent a floating-point number type;
- *mkStringSort* to represent a String type.

Two alternative techniques are instead offered by z3 to create user-defined data types, *EnumSort* and *DatatypeSort*.

An *EnumSort* allows the creation of enumerative data types, where each element of this type can be assigned a value of a predefined set (similarly as for the *enum* construct in high level programming languages like C and Java).

The Java code presented in the Listing A.1 is an example, used in the framework to create an enumerative type for the nodes of the network: *nodes* is an array of String, each one of which represents the unique identifier of the node. The created *EnumSort node* type, with name *Node*, allows to create instances which can be assigned a String value belonging to the array provided in the constructor.

```
EnumSort node = ctx.mkEnumSort("Node", nodes);
```

Listing A.1. Java code to create an EnumSort type

A *DatatypeSort* offers more richness in expressiveness, because it allows the creation of a complex data type which includes other multiple data types as fields (similarly as the *struct* construct in common high level programming languages). The Java code presented in the Listing A.2 is an example, used in the framework to create a complex structured type for the packets which flow in the network. In respect to the previous case, the steps to perform are multiple:

1. an array of String labels must be created, where each one represents the name of the internal field of the external type;
2. an array of *Sort* elements, where for each component of the user-defined type the developer wants to create its corresponding z3 data type must be provided by means of a *mkTypeSort* method;
3. a *Constructor* object must be created with the method *mkConstructor* of Context class, whose four main important parameters are, in order, the name of the type the developed is going to define, the name of the function used to identify instances of this type, the array of String labels and the array of *Sort* elements;
4. finally the *DatatypeSort* object is created with the *mkDatatypeSort* method, which accepts as input the name of the new user-defined type and an array of Constructor used for its internal representation.

```
String[] fieldNames = new String[]{
"src","dest","inner_src","inner_dest","origin","orig_body","body",
   "seq", "lv4proto", "src_port", "dest_port", "proto",
   "emailFrom","url","options","encrypted"};
Sort[] srt = new Sort[]{
address,address,address,address,node,ctx.mkIntSort(),
   ctx.mkIntSort(),ctx.mkIntSort(),ctx.mkIntSort(),
   port_range,port_range,
   ctx.mkIntSort(),ctx.mkIntSort(),ctx.mkIntSort(),
   ctx.mkIntSort(),ctx.mkBoolSort()};
Constructor packetcon = ctx.mkConstructor("packet", "is_packet",
   fieldNames, srt, null);
DatatypeSort packet = ctx.mkDatatypeSort("packet", new
   Constructor[] {packetcon});
```

Listing A.2. Java code to create an DatatypeSort type

A.1.2 Creation of z3 variables

The **Expr** class instances represent constant variables whose values do not change but they can be determined by the developer by means of hard-constraints or automatically by z3 optimizer engine through the definition of soft constraints.

To create z3 instances of in-built data types, it is possible to invoke specific methods returning a object of a corresponding subclass of *Expr* class; all these methods require an input parameter, which is a String representing the name of the variable to create. In these cases, specifying the Sort is not required because the type is inferred by z3. A selection of standard method are provided in the following:

- *mkBoolConst* to create a BoolExpr variable;
- *mkIntConst* to create an IntExpr variable;
- *mkRealConst* to represent a RealExpr variable.

It is worth noticing that the *StringSort* type does not have an immediate equivalent class for the instances. For all the *Sorts* which cannot be mapped to corresponding *Expr*, like the user-defined types, the only solution is to use the *mkConst* method, which has the following prototype:

Expr mkConst (Symbol name, Sort range)

where *name* is the name of the variable (it can be a normal String or an *Expr* variable), while *range* is the *Sort* type of which the developed wants to create an instance.
Listing A.3 provides an example where a *packet* variable and a *node* variable are created; their value is not defined in this first step, but only an identifying name is assigned. As first parameters, in this example a combination of a *DatatypeExpr*, *fw*, and of a String is provided, to show how the name of the variable can be formulated with different kind of objects.

```
Expr p_0 = ctx.mkConst(fw +"_firewall_send_p_0", nctx.packet);
Expr n_0 = ctx.mkConst(fw + "_firewall_send_n_0", nctx.node);
```

Listing A.3. Java code to create instances of user-defined types

A.1.3 Basic relational logic operators

Among the Java APIs, some method can be invoked to create formulas in the relation logic by means of traditional operators like *and*, *or*, *xor*. In the following the prototypes and the description of the most relevant ones are presented:

BoolExpr mkNot (BoolExpr a) This method creates a new *BoolExpr* object by negating the input *BoolExpr* variable.

BoolExpr mkAnd (BoolExpr...t) This method creates a new *BoolExpr* object by means of the *and* operation applied to the input *BoolExpr* variables.

BoolExpr mkOr (BoolExpr...t) This method creates a new *BoolExpr* object by means of the *or* operation applied to the input *BoolExpr* variables.

BoolExpr mkXor (BoolExpr t1, BoolExpr t2) This method creates a new *BoolExpr* object by means of the *xor* operation applied to the two input *BoolExpr* variables.

BoolExpr mkEq (Expr x, Expr y) This method creates a new *BoolExpr* object stating the the input *Expr* variables should have the same value.

BoolExpr mkImplies (BoolExpr t1, BoolExpr t2) This method creates a new *BoolExpr* object where the t2 object is considered true if the t1 object is stated as true.

A concise example is provided in Listing A.4, where the *BoolExpr behaviour* variable, representing the behaviour of a packet filter, is created by negating an *or* combination of the auto-configured rules; the mkOr method receives, in fact, an array of BoolExpr instances as input.

```
BoolExpr behaviour = ctx.mkNot(ctx.mkOr(rules.toArray(tmp)));
```

Listing A.4. Java code to show an example about how to use logic operators in z3

A couple of useful methods offered by Context class are, in addition, *mkTrue()* and *mkFalse()* methods, which respectively returns a*BoolExpr* object set to the true or false boolean value; they are fundamental, to force boolean expressions to a specific value in hard constraints.

A.1.4 Quantifiers

Quantifiers are a powerful feature which allows the specification of formulas for the set of all the instances of a specific data type; the syntax makes the work of the developer easier, because the number of formulas and variables involved is much less, but on the other hand performances are worst - this is the reason for which all the formulas involving quantifiers for nodes in the network were removed during this thesis work. Nevertheless packets are still often referred through quantifiers, for this reason this guide provides this section to describe how to use this feature.

The Java method that can be invoked to create a formula involving quantifiers is the following:

Quantifier mkForall (Sort[] sorts, Symbol[] names, Expr body, int weight, Pattern[] patterns, Expr[] noPatterns, Symbol quantifierID, Symbol skolemID)

where the most important input arguments (the others can be set to *null*) are:

sorts the *Sorts*, e.g. data types, of the variables expressed through quantifiers

names the names of the variables

body the body of the quantifier, i.e. the formula involving variables with quantifiers

weight the importance of using the quantifier during association (default value is 0)

Listing A.5 provides an example, where a formula involving a quantifier for the *packet* type is involved: basically, this formula is evaluated for every instance of this *Sort* defined in the z3 model. It is so evident how it leads to a gradation of performances, in relation to the number of instances - only for packets it is still acceptable, since their number is limited.

```
Quantifier quantifier = ctx.mkForall(new Expr[] { p_0 },
        ctx.mkImplies(ctx.mkAnd((BoolExpr) recv,
        behaviour, used), ctx.mkAnd(enumerateSend)),
        1, null, null, null, null);
```

Listing A.5. Java code to show how quantifiers works

A.2 Optimize class

The **Optimize** class is the key element for the formulation of a MaxSMT problem; in VEREFOO it has been used to solve a weighted partial MaxSMT problem. The reference class for a SMT problem is, instead, the Solver class, exploited in Verigraph; despite this, most of the methods which can be invoked on a Optimize object have equivalent methods in the Solver class (e.g. Optimize class provides the *Push()* method, while Solver class calls it *push()*).

As showed in Listing A.6, through the method *mkOptimize*, an Optimize object is created. It can later be used to add constraints and parameters before invoking the z3 optimizer engine to decide if the problem is satisfiable.

```
Optimize solver = ctx.mkOptimize();
```

Listing A.6. Java code to show how to create an Optimize object

A.2.1 How to define a MaxSMT problem instance

The workflow to define and solve a MaxSMT problem by using an Optimize object can be summed up with the following points:

1. an Optimize object is created (from this moment, it will be called *solver* as the name of the Optimize variable used in the framework);
2. a backtracking point is created with the *Push()* method;
3. the soft and hard constraints are added;

4. some parameters are setted by means of the method *setParameters*, which requests a *Params* object as input;

5. the method *Check()* is invoked to get a *Status* object representing the result of the z3 optimizer engine;

6. finally the *Pop()* method backtracks the backtracking point previously created.

Listing A.7 provides an example representing the flow of methods to invoke for the definition of a MaxSMT problem instance; it is worth noticing how in the *Params* object the proper engine is selected. The method *addConstraints()* is instead an internal method of the VEREFOO framework, in charge of adding all the hard and soft constraints on the *Optimize* object.

```
Optimize solver = ctx.mkOptimize();
solver.Push();
addConstraints();
Params p = ctx.mkParams();
p.add("maxsat_engine", ctx.mkSymbol("wmax") );
p.add("maxres.wmax", true);
solver.setParameters(p);
Status result = this.solver.Check();
solver.Pop();
```

Listing A.7. Java code to show how to define a MaxSMT problem

A.2.2 How to add hard and soft constraints

The creation of a hard constraint - that is a not-relaxable clause which the z3 optimizer engine must satisfy to provide a satisfiability result to the problem instance independently of the achieved cost - or of a soft constraint - that is not required to be satisfied, despite the engine tried to do it - is trivial, because they are generic *BoolExpr* objets defined by means of methods like the ones presented in A.1.3.

Then, in order to add hard constraints in the MaxSMT problem instance, the method *Add* can be invoked on the *Optimize* object (*Assert* is another method representing an alias of *Add*); its prototype is the following:

void Add (BoolExpr...constraints)

where it is possible to add more hard constraints at the same time.

To add of a soft constraint, instead, a specific method of *Optimize* class - the *AssertSoft* method - is required:

Handle AssertSoft (BoolExpr constraint, int weight, String group)

where the three input parameters are:

constraint It is the *BoolExpr* object representing the relaxable clause.

weight It is the weight, i.e. the importance of the constraint; the higher this value is, the bigger the priority to satisfy this constraint is, even though not strictly needed.

group It is the name of the group of soft constraints, if a partitioning is needed.

Supposing that *hardConstraint* and *softConstraint* are two *BoolExpr* objects representing respectively a not-relaxable and a relaxable clause, then Listing A.8 provides an example about how the aforementioned method can be invoked to add these constraints in a MaxSMT problem instance through the *Optimize* object here called *solver*. These methods must be invoked after the *Push()* method.

```
solver.Add(hardConstraint);
solver.AssertSoft(softConstraint, 100, "fw_group");
```

Listing A.8. Java code to show how to add constraints

A.2.3 How to get the result

The *Status* object returned by the invocation of the *Check* method on the *Optimize* object can have three different values:

1. *Status.UNSATISFIABLE*, if the optimizer engine did not find any solution for the problem (e.g. it does not exist any combination of the values assigned to the variables in the soft constraints which satisfies all the hard constraints);

2. *Status.UNKNOWN*, if the optimizer engine did not succeed in finding a solution because of external factors even though satisfiability could be theoretically reached (e.g. problems could be that the execution runs out of memory or a segment of the problem is undecidable as in some non-linear integer arithmetic expressions);

3. *Status.SATISFIABLE*, if the optimal solution to satisfy the MaxSMT problem was successfully determined by the optimizer engine.

If the result is *Status.SATISFIABLE*, then it is possible to retrieve the *Model* object invoking the method *getModel()* on the *Optimize* object, as it is shown in Listing A.9.

```
Model model = null;
if (result == Status.SATISFIABLE) {
        model = this.solver.getModel();
}
```

Listing A.9. Java code to show how retrieve the result

The method *toString()*, if applied to the *Model* object, allows to retrieve all the assertions of the z3 instance of the MaxSMT problem, so that it is possible to understand which soft constraints have been effectively satisfied; instead, if a result of satisfiability has been reached, it already means that all the hard constrains have been satisfied.

Alternatively, the most recent versions of z3 support a new method, *getAssertions()*, which, if invoked on the *Model* object, returns an array of *BoolExpr* elements representing the clauses of the MaxSMT problem.

Appendix B

RESTful APIs for ADP module

A new REST-based interface has been designed for the *ADP* (Allocation-Distribution-Placement) module to allow an interaction with all the other components of the framework and the service designer. This interface is integrated in the Spring Boot framework, which directly embeds Tomcat and does not require to deploy WAR files; it is, consequently, a versatile, user-friendly and efficient tool which can be exploited to interact with the main functionalities provided by the *ADP* module

B.1 Resource Design

Figure B.1 shows how the design of the resources identified five main areas of interest, for which information should be stored in a database:

- A *graph* resource represents an Allocation or Service Graph, identified by a long unique identifier *gid*. Each graph is characterized by a set of *nodes* resources, identified by their unique string name *nid*; other important features of a node are its neighbours, useful to model the links in the network, and the possible configuration in case it is provided by the service designer; additionally, some *constraints* can be introduced about the requirements of some functions to deploy or about the automatic generation of an Allocation Graph.

- A *requirements* resource represents a set of Network Security Requirements which can be introduced in the *ADP* module either directly by the service designer of another module, the *H2M*, of VEREFOO. Each *requirements* element is identified by a unique identifier *rid* and is characterized by a set of *properties* representing the reachability and isolation constraints.

- A *substrate* resource represents the physical network on which the VNFs can be placed; it is identified by a unique identifier *sid* and is characterized by a set of *hosts* representing the physical servers.

- A *function* resource represents a Network Security Function which the *ADP* module can allocate on the logical topology and is uniquely identified by

its name *fid* (at the current state of the art, only packet filter firewall is supported).

- A *simulation* resource represents the result of a VEREFOO run, if the outcome is positive; the corresponding Service Graph or Physical Graph, with all the information about the satisfied Network Security Requirements, is stored with a unique identifier *sid*.

B.2 RESTful APIs Design

Table B.1 shows the complete design of the Rest APIs because on the above described resources. The main principles which were followed in this operation and in the subsequent implementation by means of the Java Enterprise Edition *JAX-RS* framework are the following:

- a *singleton* class has been developed to store the data provided by the service designer in the main memory, so that they can be retrieved and combined in a single data structures on which the simulation of the *ADP* module is then performed;
- all the operations are managed to be performed in a concurrent environment, by exploiting Java data structures like *ConcurrentHashMap* and the *synchronized* keyword for the critical methods;
- every time there is a reference in the graph to an element which still does not exists (e.g. a neighbour), if this inconsistency cannot be automatically managed, then a *Bad Request* status is provided to the user.

The simulation can, furthermore, be launched in two alternative ways:

1. the service designer can introduce a *NFV* element in the Request body to ask the *ADP* module to perform the simulation on this data structure;
2. the service designer can specify a set of identifiers as query parameters in the Request, so that the related resources are automatically retrieved and, if they are sufficient to perform a simulation, it is launched.

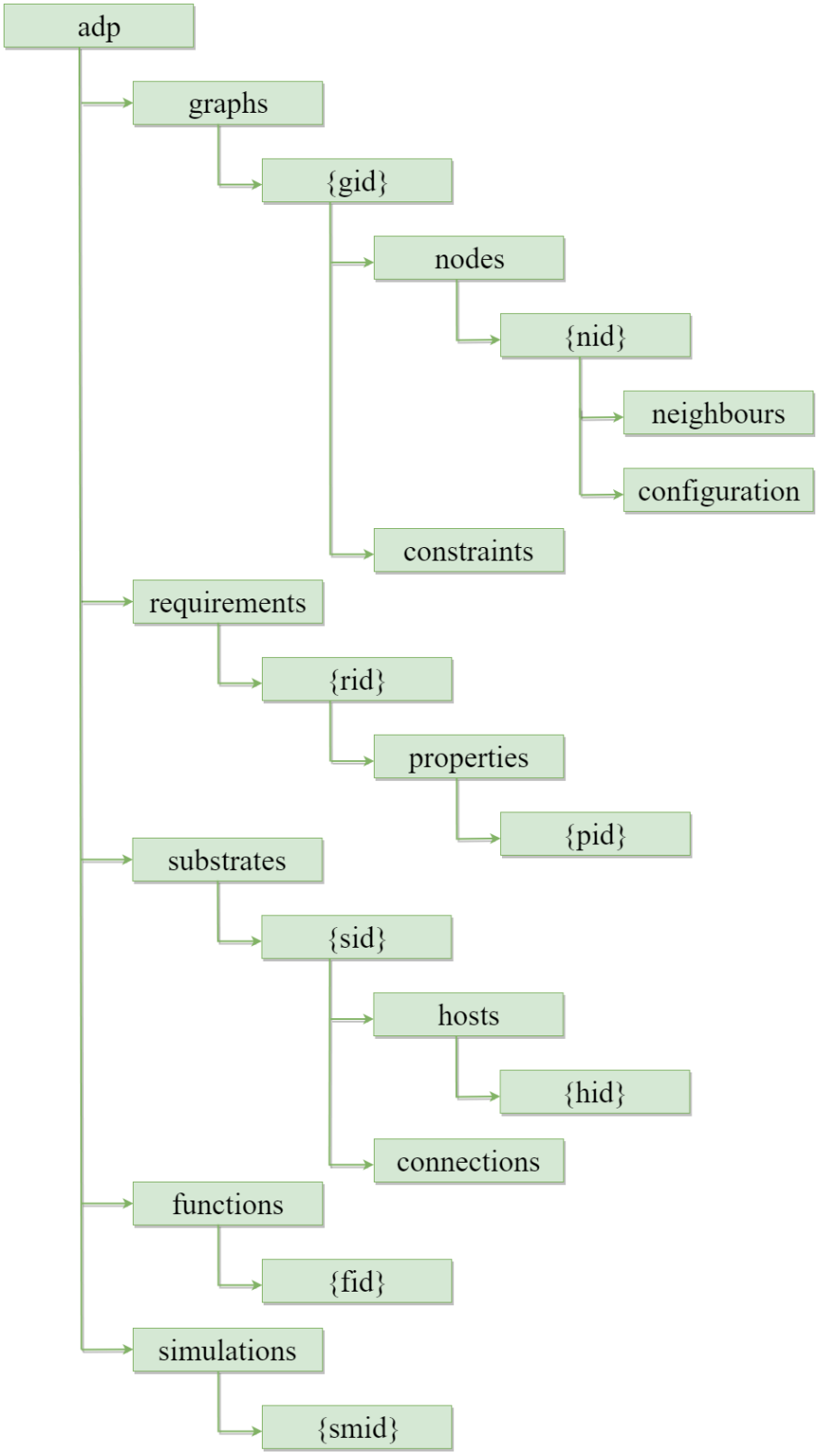


Figure B.1. Resource Design

Table B.1: RESTful API Design

Resource	*Verb*	*Query parameters*	*Request Body*	*Status*	*Response Body*	*Description*
adp/graphs	POST		Graph	201 Created 400 Bad Request	Graphs	Create a new graph.
adp/graphs	GET			200 Ok 404 Not Found	Graphs	Retrieve all the graphs in the database.
adp/graphs	DELETE			204 No Content 404 Not Found		Delete all the graphs.
adp/graphs/{gid}	PUT		Graph	204 No Content 404 Not Found		Update an existing graph.
adp/graphs/{gid}	GET			200 Ok 404 Not Found	Graph	Retrieve a graph by its ID.
adp/graphs/{gid}	DELETE			204 No Content 404 Not Found		Delete an existing graph.
adp/graphs/{gid}/nodes	POST	nid = unique name of the node		201 Created 400 Bad Request 409 Conflict	Node	Create a new node for a given graph.
adp/graphs/{gid}/nodes/{nid}	PUT		Node	204 No Content 400 Bad Request 404 Not Found		Update an existing node of a given graph.
adp/graphs/{gid}/nodes/{nid}	GET			200 Ok 404 Not Found	Node	Retrieve a node of a given graph.
adp/graphs/{gid}/nodes/{nid}	DELETE			204 No Content 404 Not Found		Delete a node of a given graph.

Resource	*Verb*	*Query parameters*	*Request Body*	*Status*	*Response Body*	*Description*
adp/graphs/{gid}/ nodes/{nid}/ neighbours	POST		Neighbour	204 No Content 404 Not Found 409 Conflict		Add a new neighbour for the node.
adp/graphs/{gid} nodes/{nid}/ neighbours	DELETE	*neighbour* = name of the neighbour to remove		204 No Content 400 Bad Request 404 Not Found		Delete the neighbour of the node.
adp/graphs/{gid}/ nodes/{nid}/ configuration	PUT		Configura – tion	204 No Content 404 Not Found		Update the configuration of the node.
adp/graphs/{gid} nodes/{nid}/ configuration	DELETE			204 No Content 404 Not Found		Delete the configuration of the node.
adp/graphs/{gid}/	POST		Constraints	201 Created 400 Bad Request 409 Conflict	Constraints	Define a set of constraints for the graph.
adp/graphs/{gid}/ constraints	PUT		Constraints	204 No Content 400 Bad Request 404 Not Found		Update the constraints of the graph.
adp/graphs/{gid}/ constraints	GET			200 Ok 404 Not Found	Constraints	Retrieve the constraints of the graph.
adp/graphs/{gid}/ constraints	DELETE			204 No Content 404 Not Found		Delete the constraints of the graph.

Resource	*Verb*	*Query parameters*	*Request Body*	*Status*	*Response Body*	*Description*
adp/requirements	POST		Property Definition	201 Created 400 Bad Request	Property Definition	Create a new security requirements set.
adp/requirements	GET			200 Ok 404 Not Found	List ⟨*Property Definition*⟩	Retrieve all the security requirements sets.
adp/requirements	DELETE			204 No Content 404 Not Found		Delete all the security requirements sets.
adp/requirements/{rid}	PUT		Property	204 No Content 404 Not Found		Update an existing requirements set.
adp/requirements/{rid}	GET			200 Ok 404 Not Found	Property	Retrieve an existing requirements set.
adp/requirements/{rid}	DELETE			204 No Content 404 Not Found		Delete an existing requirements set.
adp/requirements/{rid}/property	POST		Property	201 Created 400 Bad Request	Property	Create a new security requirement.
adp/requirements/{rid}/property/{pid}	PUT		Property	204 No Content 404 Not Found		Update an existing security requirement of a given set.
adp/requirements/{rid}/property/{pid}	GET			200 Ok 404 Not Found	Property	Retrieve a security requirement of a given set.
adp/requirements/{rid}/property/{pid}	DELETE			204 No Content 404 Not Found		Delete a security requirement of a given set.

Resource	*Verb*	*Query parameters*	*Request Body*	*Status*	*Response Body*	*Description*
adp/substrates	POST		Hosts	201 Created 400 Bad Request	Hosts	Create a new substrate network.
adp/substrates	GET			200 Ok 404 Not Found	List ⟨*Hosts*⟩	Retrieve all the substrate networks in the database.
adp/substrates	DELETE			204 No Content 404 Not Found		Delete all the substrate networks.
adp/substrates/{sid}	PUT		Hosts	204 No Content 404 Not Found		Update an existing substrate network.
adp/substrates/{sid}	GET			200 Ok 404 Not Found	Hosts	Retrieve an existing substrate network.
adp/substrates/{sid}	DELETE			204 No Content 404 Not Found		Delete an existing substrate network.
adp/substrates/{sid}/hosts	POST	hid = unique name of the host	Host	201 Created 400 Bad Request	Host	Create a new host.
adp/substrates/{sid}/hosts/{hid}	PUT		Host	204 No Content 404 Not Found		Update an existing host of a given substrate network.
adp/substrates/{sid}/hosts/{hid}	GET			200 Ok 404 Not Found	Host	Retrieve an existing host of a given substrate network.
adp/substrates/{sid}/hosts/{hid}	DELETE			204 No Content 404 Not Found		Delete an existing host of a given substrate network.

Resource	*Verb*	*Query parameters*	*Request Body*	*Status*	*Response Body*	*Description*
adp/substrates/{sid}/	POST		Connections	201 Created 400 Bad Request 409 Conflict	Connections	Define a set of connections for the substrate network.
adp/substrates/{sid}/ connections	PUT		Connections	204 No Content 400 Bad Request 404 Not Found		Update the connections for the substrate network.
adp/substrates/{sid}/ connections	GET			200 Ok 404 Not Found	Connections	Retrieve the connections of the substrate network.
adp/substrates/{sid}/ connections	DELETE			204 No Content 404 Not Found		Delete the the connections of the substrate network.
adp/functions	POST		String	201 Created 409 Conflict	String	Create a new function type.
adp/functions/{fid}	GET			200 Ok 404 Not Found	String	Retrieve a function type.
adp/functions/{fid}	DELETE			204 No Content 404 Not Found		Delete a new function type.

Resource	*Verb*	*Query parameters*	*Request Body*	*Status*	*Response Body*	*Description*
adp/simulations	POST		NFV	201 Created 400 Bad Request	NFV	Run a simulation based on a *NFV* complete element.
adp/simulations	POST	gid = id of the graph; rid = id of the requirements set; sid = id of the substrate; fid = name of the function (multiple are possible)		201 Created 400 Bad Request	NFV	Run a simulation based on a set of resources retrieved, if they exist, exploiting the input parameters.
adp/simulations/{smid}	GET			200 Ok 404 Not Found	NFV	Retrieve the result of a past simulation.