POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Thesis

Optimal configuration of security controls in software networks

Supervisor
Prof. Cataldo Basile
Prof. Antonio Lioy
Dr. Fulvio Valenza

Candidate
Andrea Della Chiesa

Academic Year 2019-2020

To my grandmother

Summary

Nowadays, words like cloud-computing, cyber-security and datacenters are often used as keywords to define the modern Internet technologies. In the last decades, we assisted to a rapid growth in the use of the Internet, both for business and private users. Nonetheless, the migration to the "connected world" nhas brought new research topics and technology innovations. yt9 By using virtualization techniques, it is possible to abstract the physical resources and assign them dynamically to the users. This has allowed the development of the Network Function Virtualization (NFV) which has been empowered with Software Defined Networks (SDN) technologies. Creating virtual machines, attaching them to virtual networks, replicating or migrating them are easy operations that can be done in software networks. Moreover, deploying network services became faster and intuitive, ensuring high availability and fast provisioning.

Never like in this period we assisted on how the Internet technologies are fundamental to provide remote business services, maintain contacts with other peoples or to give important communications and informations. Along with the growth of these technologies, the risks associated to cyber-attacks became not negligible. Cyber-security is one of the central themes nowadays, and several business companies has focused its researches on this topic.

Cloud-security became fundamental to ensure the protection of the cloud environments, it has to implement optimized security mechanisms to provide a fast reaction to each possible attack. Data protection it is one of the most important task in a cloud environment, because on these years, the amount of private or business sensitive informations that flows in the datacenters has grown exponentially.

These systems, however, do not provide the same elasticity and optimization techniques when security requirements need to be enforced. It is therefore important to define new methods and tools to support administrators in security-related tasks. Such results could extend the benefits and the flexibility of the modern Internet technologies also to security, providing fast, optimized, and on-demand security services.

This thesis work aims to provide an automatic security controller, employable in various cloud infrastructures, which optimizes the distribution of the firewalls filtering rules over the cloud environment.

Acknowledgements

During this five-year journey I have enriched my technical background, but most of all I have understood what are my interests and objectives for the future.

I would like to acknowledge my supervisors, Prof. Basile and Prof. Lioy whos gave me the possibility to work on interesting research matters, in a productive and stimulant environment. Moreover I would like to acknowledge Dr. Valenza for his continuous presence during the research work, helping me in technical and practical issues.

Then, I would like to acknowledge my parents, who gave me the possibility to realize my objectives. During these years they always believed in me, supporting me in every moment of this journey. I would like to thank my brother, who is my wingman in every moment of my life. Finally, my grandparents and all my relatives, whose support was always present.

I want to acknowledge Igor, for all the moments we had shared and for its contribution in this thesis work.

A special thank to my friends, who always included me, despite the distance between us. Last but not least, I want to thank my roommates, with whom I have shared beautiful moments during these years.

This is the end of a beautiful path, I have faced difficult moments, but most of them led to great successes. I will never forget this period of my life, and I know that this is the starting point for a new chapter of my life.

Contents

Chapter 1

Introduction

Internet has became one of the central themes of the modern era, *Cyber security*, *Cloud computing*, *datacenters*, are some of the most common words used in the last years. These technologies, nowadays, are the central core of most of the business activities around the world and the number of services offered is growing constantly.

According to Statista, in 2018, the public cloud computing market is projected to be worth around 141 billion U.S. dollars [28]. As shown in Figure 1.1 the market has seen massive growth over the past decade, skyrocketing from a value of less than six billion dollars a decade ago. By 2020, the projected increase in cloud adoption will reach almost 240 billion dollars.

Along with this rapid and fast growing expansion of cloud services, the necessity of powerful infrastructures, computing devices and storage systems became unavoidable. Moreover, every provider must ensure a certain level of security to its customers, in terms of data protection, service availability and identity management respecting users privacy rights.

The aspects of this extraordinary evolution brought to the research challenging problems and interesting topics. Technologies like Software-Defined-Networks (SDN) and Network-Function-Virtualization (NFV) has a central role in the development of efficient and optimized services to cope with the ever-growing traffic. These technologies exploits virtualization techniques to provide network functions and services, with all the advantages of the virtualized environments such as scalability, replication, fast provisioning and optimized resource allocation.

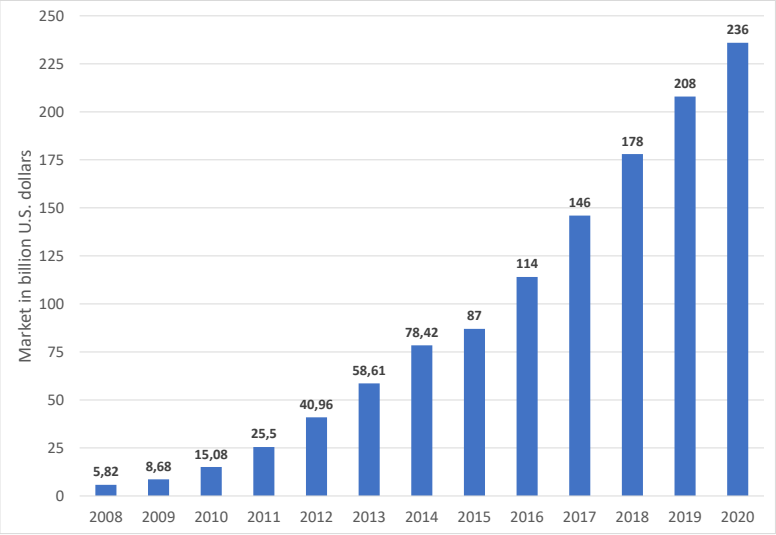

Figure 1.1. Total size of the public cloud computing market from 2008 to 2020 [28]

Cyber-security nowadays is a trending topic, companies has invested billions to improve the security of its services due to the insane amount of data and traffic that it is necessary to manage to provide a market competitive service. Moreover, the number of attacks directed to cloud services is growing as fast as the cloud expansion.

Attacks to the cloud infrastructures could damage business, but also private users. Data loss or leakage, could be catastrophic for companies and it could compromise their business activities. Denial-of-Service (DoS) attacks aim to damage the service availability of a provider damaging everyone is using the services. To overcome these problems it is necessary to provide fast, easy, on-demand security services that can be utilized in every environment at any time.

The *SECaaS* (Security-as-a-Service) paradigm fits these requirements, providing a rapid security service to the customers. It offers, in a cloud environment, a series of customizable security services. The customer can exploit the *pay-per-use* approach using only the services needed for its business requirements.

These systems, however, do not provide the same elasticity and optimization techniques when security requirements need to be enforced. Often, network administrators have to manually configure the security devices, which is known to be an error-prone task. Indeed, several analyses demonstrated that, in most of the cases the cyber-attacks are caused by human errors. Moreover, due to the rapid migration and relocation mechanisms provided in software networks, manual reconfigurations of security control need to be done too frequently.

Firewalls and Intrusion-Detection-Systems (IDS) became fundamental for traffic filtering and attacks prevention, but with the growth of the amount of traffic that needs to be processed, they has to be powerful and fast enough to handle it.

This thesis work aims to provide an automatic security controller, employable in various cloud infrastructures, which optimizes the distribution of the firewalls filtering rules over the cloud environment. The application will perform an optimization over several security policies and, according to the pattern of traffic targeted by the filtering rules, it perform an optimized distribution of them.

To achieve these objectives it is necessary to understand the infrastructure of a cloud environment and its applications (e.g. OpenStack, Kubernetes), and using them to build an optimization and distribution model.

In this work the distribution aspects are highlighted, studying how the infrastructure relay on SDN and NFV technologies and how to exploit them to perform the optimized distribution. Once identified the type of the available filtering points in the infrastructure and how to use them, I've implemented the distribution process.

For each type of filtering point, it is necessary to study its technology and how to apply security policy on them. The next phase consists in the distribution process: given a filtering policy and a target on which inject the rules, the tool will automatically load the filtering rules needed, using the proper commands or protocols.

The application will follow the *SECaaS* approach providing a full compatibility with several cloud software. Moreover the objective is to support the cloud providers in the security optimization, getting high performances without the necessity to improve the physical devices specifications.

To prove the effectiveness of the optimization, several performance tests are done highlighting the difference between the standard security mechanisms provided by standard cloud toolkits (e.g. OpenStack) and the realized optimization tool. The metrics used to prove the efficiency of the tool were the network bandwidth and latency and the overall CPU consumption.

The tests highlights the performance gain of the policy optimization due to the reduction of the number of rules. They showed that an optimized policy, without anomalies or redundancies, could reduce the overall number of rules that need to be injected on the devices, with a consequent increase in performance. Moreover, the optimized distribution process leads to a significative improvement, loading the rules only on the devices interested by the traffic pattern they identify.

This thesis work is divided in the following chapters and topics.

- **Chapter 2** presents an introduction to the concepts used in this work (Ch. 2),
- **Chapter 3** is an overview of the current *State-of-the-art* and which are the current research topics (Ch. 3),
- **Chapter 4** defines the goals of the project (Ch. 4),
- **Chapter 5** contains analysis of the infrastructures and components used in this work (Ch. 5),
- **Chapter 6** introduces how the solution is designed (Ch. 6),
- **Chapter 7** presents the implementation of the project (Ch. 7),
- **Chapter 8** shows the results of the testing activities (Ch. 8),
- **Chapter 9** resumes the conclusions and introduces an overview of the possible future works (Ch. 9),
- **Appendix A** is the programmer manual and contains the instruction to extend the functionalities (Appx. A),
- **Appendix B** summarizes all the available REST API (Appx. B).

Chapter 2

Background

In this chapter there is an overview of the main technologies used in this thesis work.

2.1 Computing virtualization

Virtualization is the process of running a virtual instance of a computer system in a layer abstracted from the actual hardware.

Applications that run on virtualized Operating Systems (OSs) have the same behavior as they runs on a dedicated machine. These virtual systems are called Virtual Machines (VMs).

Virtualization can be used by everyone, from private users that want to run applications designed for a different OS without installing it, to big companies that want to consolidate different systems on a single hardware device.

This consolidation process lead to an efficient use of the hardware resources, avoiding to use an entire, powerful machine for a single low-consuming service. Therefore, the power consumption of a machine that runs several VMs inside, is lower than having a single machine per application.

We can define two types of operating system:

- Guest OS, is the OS that runs on the VMs, it should not be aware of running in an virtualized environment;
- Host OS, is the OS that includes the Hypervisor and it is responsible of hardware virtualization and VMs management and orchestration.

The hypervisor is the software that lay between hardware and VMs. It is responsible of the orchestration and the management of the Virtual Machines.

Hypervisors can be *Type-1* or *Type-2*, the former is installed directly on the underlying hardware, the latter is a normal application that runs on a Operating System (e.g. VirtualBox). Today there's another type of hypervisors that uses technologies like Kernel-based Virtual Machine (KVM) to virtualize systems using directly the underlying hardware, but running as a normal application installed in the OS.

Hypervisor have to ensure complete isolation between VMs, because a fault in a VM should not compromise the functionalities of the others. Moreover hypervisor has to be secure because a security issue in the hypervisor can make vulnerable all the VMs that runs on it.

Virtualization techniques bring several advantages in terms of agility, consolidation and isolation.

- Agility: virtualization extend VMs controllability, in fact is possible to pause/restart it, migrate it on another host, duplicate it or change the amount of resources assigned to it. Moreover with virtualization is possible to add/remove peripherals, simulating them without touching the real hardware.

- Consolidation: aggregating VMs into a single physical server can reduce the overall energy consumption. It optimizes the utilization of the physical resources avoiding to have multiple servers that runs a single application.

- Isolation: critical application can be isolated in different VMs avoiding that malicious actions on a VMs can interfere with applications running on other VMs.

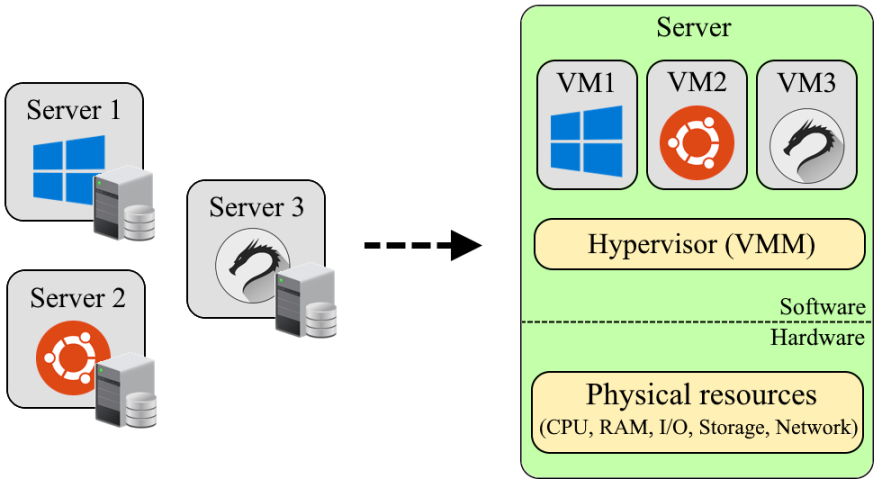

Figure 2.1. Computing virtualization

Figure 2.1 shows how a set of physical servers can be instantiated on a single physical device with different virtualized environments.

Nevertheless, virtualization is computationally expensive and it can be difficult to obtain high performance in an virtualized environment. Some solution was introduced (e.g. hardware offloading) but achieving high performances is still one the main challenges.

2.1.1 Network virtualization

Virtualization introduces additional requirements for networking because is necessary to guarantee the communication between VMs and between the physical network and the virtualized machines. It may be also necessary to assign different IP addresses to the VMs and configure additional network services (e.g. firewall, load balancers, etc.).

The main idea is to create a virtual topology that acts as the physical one. The VMs can be connected at Ethernet (L2) or IP (L3) level.

- L2 connection: connecting VMs at Ethernet level can be done with a virtual switch, implemented in software, that runs inside the physical server. Alternatively the connection can be done by the physical Network Interface Card (NIC) or by an external switch connected to the server.

- L3 connection: there are two possible implementation to provide L3 connection between VMs: native networking and overlay networking.

  – With Native networking the IP addresses of the VMs are given according to the underlying network. It may be necessary to introduce VLANs to ensure communication between VMs. For this approach is necessary the cooperation of the network provider to guarantee the connectivity.

  - Using Overlay networking the IP addresses of the VMs are hidden to the network infrastructure. The physical network is decoupled from the virtual one and it is not necessary to add specific routes for VMs communication. If VMs are running in different physical hosts it may be necessary to implement tunneling mechanism (e.g. GRE, VxLAN).

SDN technologies can improve the performances of virtualization ensuring fast and reliable connections between VMs and providing *context-based* traffic forwarding. Moreover, these technologies decouples data plane from control plane, centralizing into a single controller the entire network management.

2.2 Cloud computing

Virtualization techniques provide elasticity in resource assignment. With this approach machines no longer need particular characteristics of performances (e.g. CPU, RAM, Storage, etc.) because resources are assigned depending on the actual need.

Customers can buy a large number of servers with same performances and hardware specification and consolidate them in a single datacenter, using virtualization to implement their services. Commercial-Off-The-Shelf (COTS) are products that are ready-made and available for sale to the general public, they are widely used to implement cloud architectures and datacenters.

When public datacenters came to the market the concept of Cloud Computing started to appear. According to National Institute of Standards and Technology (NIST), Cloud Computing can be defined as: "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [6].

With cloud computing users can buy exactly the amount of resources they need, with a *pay-per-use* approach. Moreover, the user can be "unaware" of underlying technologies and infrastructure.

Elasticity is one of the main advantages of cloud computing, resources can be scaled automatically and dynamically, according to the actual needs. Cloud Provider can also offer *geo-localized* services on their datacenters around the world, optimizing latency and response time. Users can specify preferences about data localization, but cloud provider does not provide informations about the exact location of the provided resources.

Data availability and reliability is enforced with cloud solutions, ensuring redundancy, replication of data and business continuity with *always-on* servers. Services can be reached from anywhere on every kind of device via standard Internet connections.

Cloud services deployment can be divided in four models.

- Private, all the physical infrastructure is property of the user. Is up to the user implementing security, power, locals and machines, moreover it has to keep applications updated and sort possible problems.
- Dedicated Hosting, customer uses hardware, locals, internet connection, buildings and electrical power offered by the provider to deploy his services. This approach is useful for remote computation of resources expensive programs
- Public, this model abstracts the final user to the infrastructure. All the operations aforementioned have to be performed by the cloud provider.
- Hybrid, is possible to mix the previous approaches having physical and virtual server running together according to the customer needs.

Cloud computing requires a reliable and performing Internet connection to work properly. In fact, without Internet is impossible to access to public cloud features, moreover cloud applications

are not appropriate for "low-latency purposes" due to Internet latency. Users, using cloud features, partially lose control over their data. Knowing the exact location of data, and who is the legal owner of the data may be a not negligible problem. Another issue may be what it is necessary to do with data after the end of cloud provider operations and how data destruction has to be done.

2.2.1 Serivce models

Cloud computing architectures are defined using a series of service models, according to the customer needs. It is possible to buy the physical machines and implementing all the infrastructure, or even buy a complete *ready-to-use* cloud application.

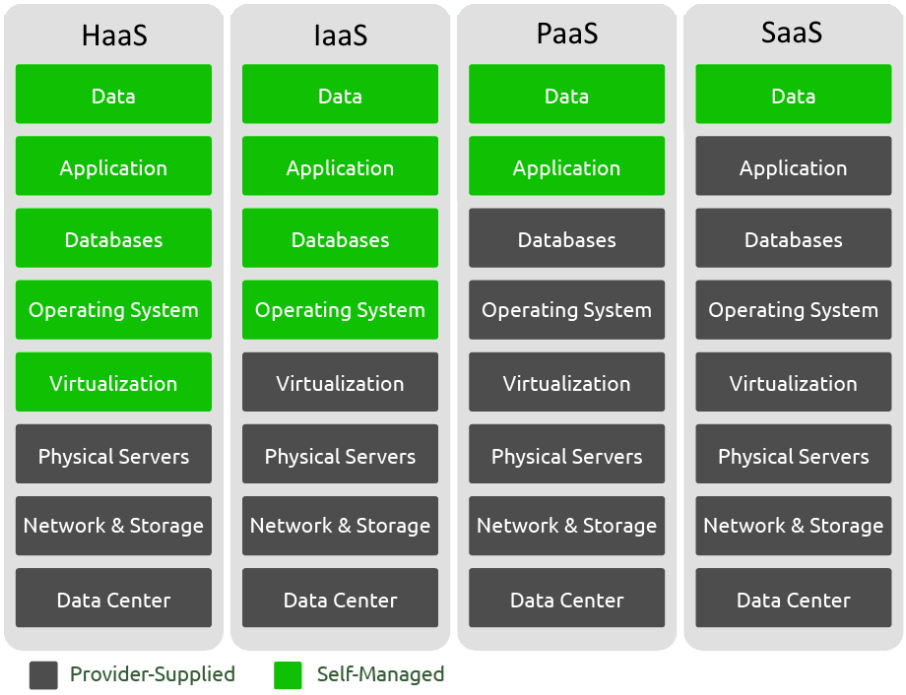

Figure 2.2. Cloud computing service models

As shown in Figure 2.2, these services models have been grouped into four main classes.

- Hardware-as-a-Service (HaaS): the cloud provider offers all the physical hardware needed by the customer. It provides locals, electrical power, air cooling systems, internet connection, physical access security, etc. It can be useful for remote computation, when the user's application needs high computational resources, or for security reasons. The customer has to manage the network configuration, load-balancers, VMs, databases, etc.

- Infrastructure-as-a-Service (IaaS): the customer can purchase almost unlimited instances and virtual machines. Usually clients can manage their VMs via Application Programming Interfaces (APIs) or via web dashboards. The cloud provider heavily exploits virtualization mechanisms to create execution environments identical to the physical ones. Users can create databases and connect them to the VMs via virtualized networks. The customer is responsible of the security of his applications because a weakness in an user application may expose VMs and the underlying infrastructure to unauthorized accesses. Vendor have to ensure service availability according its Service Level Agreement (SLA) with the customer.

- Platform-as-a-Service (PaaS): Cloud Platform Services provide components to design custom software independently from the underlying OS and infrastructure. Customers can build their *middleware* without worries about scalability, reachability and traffic balancing,

all these detail are hidden to the user. With PaaS there is a complete interaction with databases, messaging-services and other accessory services that allow them to implement a complete cloud application in few lines of code. Nevertheless, PaaS can bring some problems of compatibility with certain programming languages, moreover legacy systems may have integration problems and they may need to be customized. There is a problem known as "vendor lock-in" because technologies and implementations used by a provider may be not compatible with another vendor. If the vendor has not provisioned migration policies, switching to alternative PaaS may not be possible without affecting the business.

- Software-as-a-Service (SaaS): cloud providers can offer complete cloud application to the users. They don't need to write lines of code or to think about software compatibility and maintenance. Vendors have to ensure service availability and reliability, but also data protection and consistency. With this approach, software installations are no longer required, moreover vendor has to offer technical assistance to the customers. These *ready-made* services can be accessed over the Internet, simplifying their utilization. Cloud services can suffer a lack of features with the respect of the "client" version of the software. Moreover, integrating cloud application with existing architecture may be an hard job. With SaaS, the problem of vendor lock-in is still present because not every provider follows standard APIs and protocols, so migrating to another vendor is not an easy task. Security is an important key point of SaaS, in fact, using cloud services, users agree to the transfer of sensitive business information in the backend datacenters network, this rise new security issues that need to be handled.

2.2.2 Cloud security

Security is one of the topmost concerns of any computing model and cloud computing is not an exception. Cloud security focuses on what measures is necessary to adopt to ensure secure services and architecture to customers. To implement secure services is necessary to know cloud infrastructures, possible configurations, how the service is provided, where are data stored and technologies used to store them.

Cloud security is a fact that concerns the provider, but also the customer: the former must ensure security of its own infrastructure as well as of the clients data and applications, the latter must verify and make sure that the provider has employed all possible security measures to make the services secure. For this reason a Service Level Agreement (SLA) between Cloud Service Provider (CSP) and the customer underline the main security aspects of the cloud solution. Customers have to explain clearly what security requirements they have.

In cloud environment there are multiple threats that have to be addressed in order to ensure a reliable and secure service. Most of the threats concern about customers' data location, utilization and protection. There are also threats associated to account hijacking and identity theft, which have to be addressed by the customer enforcing his account access methods. Multi-tenancy cloud environments that exploit virtualization techniques have to ensure isolation between users avoiding possible *VM-to-VM* or *VM-to-VMM* attacks.

Vendors have to ensure also a complete destruction and *sanitization* of users' data after the end of service. This practice prevents malicious data recovery attempts, avoiding sensitive information thefts.

Cloud applications, have to be redesigned to implement native security functions. The paradigm of *defense-in-depth*, shown in Figure 2.3, represent the typical architecture of an application that follow security principles. It consists in different consequent levels of protection, so an attacker has to break through these layers to get access to the resources. It is necessary to have a separation of privileges and duties across employers and to set-up a control an monitoring mechanism to guarantee internal access security.

Implementing security may be challenging in cloud environments, for this reason the concept of SECurity-as-a-Service (SECaaS) was born. With SECaaS customers can offload to the cloud provider the implementation of most of the security aspects needed. This approach has several advantages such as, elasticity, fast-provisioning, scalability and continuous software updates. Therefore customers don't have to implement and maintain all the security aspect of their deployments, manage log files or keep monitoring activities in the datacenter.

SECaas covers most of the security necessities of a typical cloud environment such as: identity and access management, data loss prevention, web and email security and encryption. It also follow the *pay-per-use* approach.

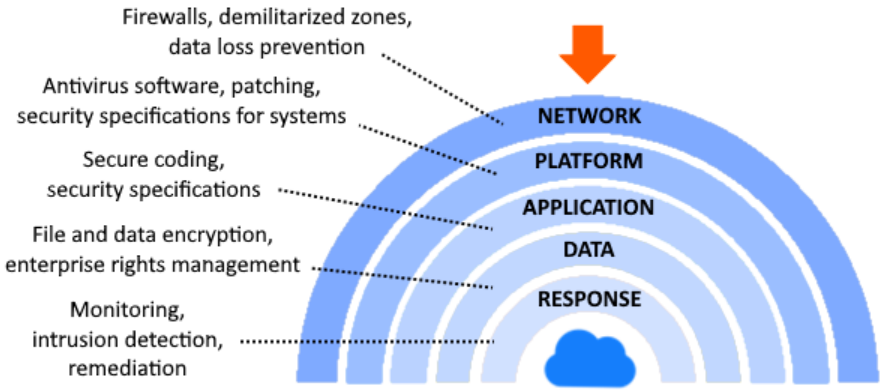

Figure 2.3. *Defense in depth* reference scheme

Cloud providers have to ensure complete availability for their services, because every downtime of a server may potentially lead to a damage to the business continuity of the customer. Building multi-tier architectures provides resiliency against Distributed-Denial-of-Service attacks.

2.2.3 Cloud toolkits

Virtual machines provide elasticity and flexibility, but managing several VMs may become a problem. Performing actions like migrating the VM, start and stop different VMs or splitting the infrastructure across multiple tenants in large datacenters with hundreds of servers can become unfeasible.

With cloud toolkits is possible to have a global view of the datacenter. The cloud provider is able to instantiate VMs, migrate VMs across different hosts, set up security features or split the underlying infrastructure with multiple tenants.

These applications provide an higher-level of abstraction of resources and allow vendosr to consolidate into a single application the complete control of the infrastructure. A cloud toolkit can be defined as an operating system for the cloud infrastructure.

A cloud toolkit can communicate with different and heterogeneous technologies such as the hypervisor, the storage system and the network resources.

- Hypervisor is responsible to orchestrate and manage the *life-cycle* of the VMs in the datacenter. It is responsible of virtualizing physical resources and manage access to them by the VMs. Hypervisor realizes also monitoring functions, sending real-time data to the cloud infrastructure and performing resource scaling as needed.

- Storage system is necessary to store data about tenants, configuration and VMs virtual disks. It's up to the provider ensure data redundancy and availability.

- Network resources ensure *VM-to-VM* communication, but also to allow VMs to communicate with Internet destinations, and, in some cases, to let VMs being reachable from outside the datacenter.

Toolkits often provides an easy and intuitive Web interface (WebUI), besides the classic Command Line Interface (CLI). These web application usually exploit REST APIs to communicate with underlying infrastructure and realize all the functions required by a datacenter.

Most of these applications are open-source (e.g. OpenStack, Apache CloudStack, Eucalyptus, etc.), but there are also proprietary services (e.g. VMWare).

2.3 Firewalls and policies

In this section packet filters and security policies are introduced.

2.3.1 Packet filters

With the expansion of Internet technologies and infrastructures across the world, network security became a key point in the research and industrial communities.

The number of threats and possible attacks is increasing continuously, therefore firewalls became indispensable in all kind of infrastructures, from enterprise networks to home network.

A firewall is a network element that controls the traversal of packets across the boundaries of a secured network based on a specific security policy [9]. It is responsible of filtering packets that don't respect filtering rules defined according to the security policy requirements.

These rules, forming an ordered set, are composed of matching fields (e.g. IP addresses, protocols, ports) that leads to an action that has to be executed on the matching packet. This action can be *deny* if the packet has to be discarded or *allow* if the packet meets the requirements to enter in the network. If a packet matches a rule, the corresponding action is executed, otherwise it is executed the *default action* which can also be *allow* or *deny*. In the following Table 2.1, a typical firewall packet filtering policy, is represented.

	Priority	Source IP address	Source port	Destination IP address	Destination port	Protocol	Action
r1	1	130.162.0.1	*	*	80	TCP	ALLOW
r2	2	130.162.0.0/24	*	*	80	TCP	DENY
r3	3	130.162.0.1/24	*	130.162.0.1/24	*	TCP	ALLOW
r4	4	130.162.0.1/24	*	130.162.0.1/24	*	UDP	ALLOW
r5	5	*	*	130.162.3.1	*	*	DENY
r6	6	*	*	130.162.3.0/24	0-1024	TCP	ALLOW
r7	7	*	*	130.162.3.0/24	*	*	DENY
...							
default	∞	*	*	*	*	*	DENY

Table 2.1. A typical firewall packet filtering policy

Rules matching is a computationally expensive task because, often, the default action of a security policy is *deny* and rules are written as exception. The decision process could be slowed down because it is necessary to evaluate an high number of rules before performing the action. Therefore is important to check if there are some rules that are in conflict to resolve them allowing packets to traverse a smaller number of rules.

To understand rules conflicts is necessary to classifying all the possible relations between rules. According to the H. Hamed and E. Al-Shaer work about the taxonomy of policy conflicts [10], it is possible to identify four conflict types.

- Exactly matching rules ($R_x = R_y$): Rules R_x and R_y are *exactly matched* if each field in the two rules is equal.

- Disjoint rules ($R_x \not\bowtie R_y$): Rules are *completely disjoint* if every field of R_x is neither a subset nor a superset and not equal to the corresponding fields in R_y. Given two fields of R_x, if one is a subset, superset or equal to the corresponding field of R_y and the other is not a subset, superset or equal to the corresponding field of R_y, the two rules are defined as *partially disjoint* [11].

- Correlated rules ($R_x \bowtie R_y$): Rules are *correlated* if at least one field of R_x is a subset of the corresponding field in R_y and the rest of the fields of R_x are supersets of the corresponding field in R_y.

- Inclusively matching rules ($R_x \subseteq R_y$): Rules are *inclusively matched* if they are not *exactly matched*, and every field of R_x is a subset or equal to the corresponding field in R_y. In this case R_y is defined as *superset match* and R_x as *subset match*.

If rules are not *completely disjoint* a packet may match more than one rule. In this case rules can be defined as *dependent* and their order must be preserved for the firewall policy to operate correctly [12]. Two *dependent* rules that have different actions, gain a different level of *precedence* depending of rule position inside the firewall. Therefore, is important to preserve rule ordering to avoid that some packet may be discarded or accepted in contrast with the original security policy.

To solve these problems it may be useful to implement optimization algorithms that can improve firewall performances. These algorithms focus mainly on *dynamic rule-ordering optimization* [12], and on *dynamic optimization of packet matching* [13].

Thanks to these optimizations, is possible to realize *ad-hoc* rules that match most of the packets that will be discarded. Then, these rule may be put on top levels of firewall tables, avoiding to traverse a large number of rules and gain high performances.

For each policy a *resolution strategy* is defined, it describes which rule action has to be executed if a packet matches more than one rule. Some standard resolution strategies are:

- First Matching Rule (FMR): the first matching rule action is executed.

- Allow Takes Precedence (ATP): if two matching rules have conflicting actions, the *allow* action is chosen.

- Deny Takes Precedence (DTP): if two matching rules have conflicting actions, the *deny* action is chosen.

- Most Specific Takes Precedence (MSTP): the rules that matches more packet fields is chosen.

- Least Specific Takes Precedence (LSTP): the rules that matches less packet fields is chosen.

There are several implementations of firewall policies, but the most common and used has *deny* as *default action* with *FMR* as *resolution strategy*. Matching fields are usually Source/Destination IP address, Source/Destination Port and Protocol, they are known as the *5-tuple*.

2.3.2 Policies

A security policy specify the security requirements that the system must satisfy and the threats it must resist [14]. A policy can contain a large number of filtering rules and if they are not *completely disjoint* may bring to several *policy anomalies*. These anomalies may occur in the same device (*intra-policy* anomalies) or between different devices (*inter-policy* anomalies).

Intra-policy anomalies

Intra-policy anomalies occurs when a packet matches two or more rules in the same firewall security policy. According to the F.Valenza and M.Cheminod work about the firewall anomalies resolution strategies [15], these kind of anomalies can be divided in *sub-optimization* anomalies and *conflict* anomalies:

- Sub-optimization anomalies: this kind of anomalies arise when there are redundant or irrelevant rules in the policy, so is possible to optimize the security policy by removing rules or by editing them. These anomalies can be divided in four sub-categories.
  - Intra-policy Exception: this anomaly occurs when a following rule is a superset of a preceding rule and its action is different. This situations often happen when is necessary to make exclusion from a general filtering rule. The problem is that some accepted traffic may be blocked from a following rule.
  - Intra-policy Redundancy: if a packet matches two rules that have the same actions, these two rules are redundant. Rule R_x is redundant to R_y if R_x precedes R_y and R_y *inclusively matches* R_x and the two rules have similar actions. This anomaly can increase the total number of rules in a security filtering policy, reducing the overall performances of the firewall.
  - Intra-policy Duplication: it is similar to redundancy but the two rules have to *exact match*. These rules match the same packet and have the same action, so the removal of one of these rule will not change the policy behaviour.
  - Intra-policy Irrelevance: according to the E. Al-Shaer and H. Hamed work about the the policy anomalies in distributed firewalls [11], this anomaly occurs when a rule do not match any traffic that flows in the firewall. For example, if a rule contains unreachable IP Addresses in the *Source/Destination IP Address* fields, this rule is *irrelevant*. This rule should be removed from the policy because it causes an unnecessary processing and a consequent decrease in performances.
- Conflict anomalies: occur when two rules match the same packets, but the respective actions are different. Rule ordering became crucial for the security policy, because the first rule that matches the packet executes its actions. If not resolved this anomalies may lead to allowing unwanted traffic or deny some authorized packets. Unlike *sub-optimization* anomalies, policy *conflicts* have to be resolved by the network administrator, because it is not possible to compute automatically the right filtering decision. These anomalies can be divided in three sub-categories.
  - Intra-policy Shadowing: this anomaly occur when a rule matches all the packet of a following rule with a different action. The shadowed rule will never have an effect in the policy. Rule R_y is shadowed by rule R_x if R_x precedes R_y and R_y *inclusively matches* R_x and the two rules have different actions. It may ne useful, in case of this kind of rule relation, to put the superset rule after the subset one.
  - Intra-policy Correlation: if two rules are correlated and their actions are different, rule ordering became crucial for the outcome of the security policy. In this case, some packets match both rules, but the respective filtering actions are different. The user have to choose the rule order to solve this issue.

– Intra-policy Contraddiction: arises if two rules *exactly* match, but their actions ar different. Is not possible to automatically remove one of the rule because their action are different, therefore is necessary that the network administrator has to resolve this conflict manually.

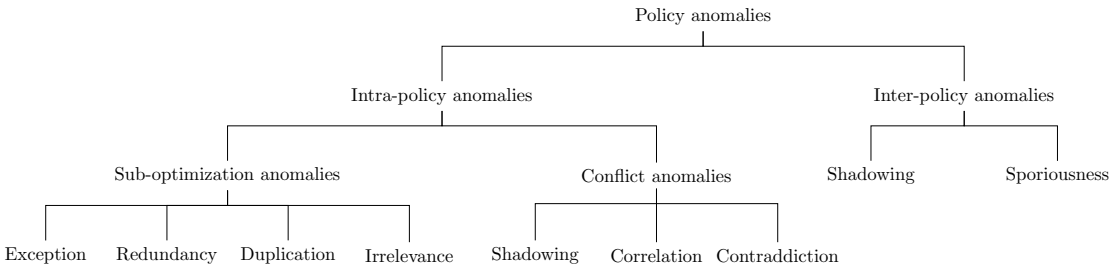

Figure 2.4. Taxonomy of policy anomalies

In Table 2.4, a hierarchical representation of the principal policy anomalies, is shown.

Inter-policy anomalies

When the security requirements affect multiple domains, it is necessary to ensure the protection using multiple filtering devices. These devices may be managed by different network administrator, so guarantee a certain level of security may result a difficult operation. Devices in the path of a flow that preceding other devices are called *upstream device*, whereas devices following are called *downstream device*. The device that is located closest to the flow destination is called the *most-downstream device*, the device closest to flow source is called the *most-upstream device* [11].

With these types of network architecture, checking only the presence of *intra-policy* anomalies is not enough to ensure proper operation. Conflict could exist between policies of different firewalls. These conflict can occur when a downstream device block traffic permitted by an upstream device and viceversa. These anomalies can be divided in two main categories.

- Inter-policy Shadowing: this anomalies arise when an upstream policy discard all or some of the traffic allowed by a downstream policy. If the downstream rule R_d and the upstream rule R_u *inclusively* match we have *partial shadowing*, we have *complete shadowing* if the R_d and R_u *exact* match. This anomaly is a problem, because some desired traffic may be dropped from upstream devices.

- Inter-policy Spuriousness: this anomalies occur when an upstream policy allows all or some of the traffic blocked by a downstream policy. If the downstream rule R_d and the upstream rule R_u *inclusively* match we have *partial spuriousness*, we have *complete spuriousness* if the R_d and R_u *exact* match. *Spurious* traffic, therefore, is authorized to travel across the network until it reach the downstream device that will discard it. Inter-policy spuriousness is a critical anomaly in network security, because it expose the underlying network to unwanted traffic and potential attacks (e.g. DDoS, Port Scanning, etc.) [10].

Identifying the anomalies between policies is a key point for a correct and efficient network security administration. There are automatic tools that can resolve some of the conflicts, but in some case human intervention is unavoidable.

Policy representation

To better understand policies and the behaviour they assume when more policies are put together, it may be useful to adopt a intuitive policy representation.

Referring to the C. Basile, A. Cappadonia, A. Lioy work about the policy transformation techniques [16], we can use a geometric model to represent security policies.

This model is composed by several elements.

- Rules $r_i = (c_i, a_i)$ can be as a *condition clause* c_i and an *action* a_i.
- Actions a_i represent the behaviour that a firewall needs to follow when a packet match the corresponding rule. The action can be *accept* if the packet can pass through the device or *deny* if it necessary to discard it. Actions are organized into an *action set* $\mathcal{A} = \{A, D\}$.
- Condition clause $c_i = s_1 \times s_2 \times \ldots \times s_m \subseteq S_1 \times S_2 \times \ldots \times S_m = \mathcal{S}$ is a subset of the selection space $\mathcal{S}$. The condition clause generate an *hyper-rectangle*.
- Selection space $\mathcal{S}$ is formed by the Cartesian product of *selectors* S_i. A selector represent the field that the packet will match (e.g. Source/Destination IP Address, HTTP Contents, etc.).
- Packets x_i can be mapped to the selectors or to the decision space. A packet match happen if the packet mapped to the selector of the condition belongs to the condition.

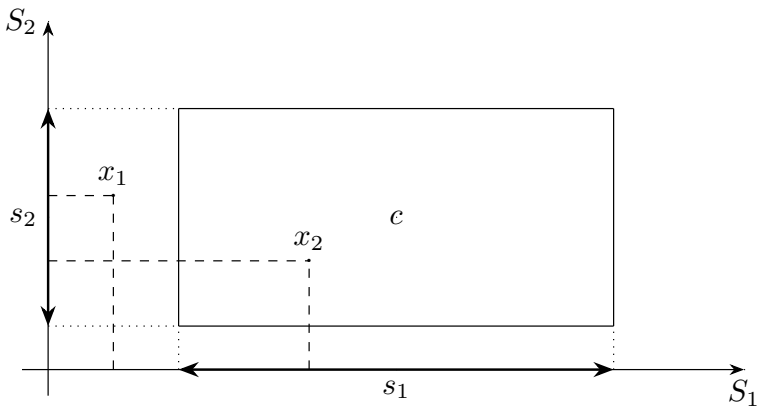

Figure 2.5. Geometric representation of a policy rule

For example, according to Figure 2.5, packet x_1 is not matching the policy rule, whereas packet x_2 mapped on the selectors S_1 and S_2 belongs to the condition c.

In this geometric model a policy can be expressed as a four-tuple $(R, \Re, E, d)$.

- $R = \{r_i\}_i, i \in [1, n]$ is a set of n rules that compose the policy.
- $\Re : 2^R \to \mathcal{A}$ is the *resolution strategy*, it determines the action to execute if a packet matches more than one rule. For example, *First Matching Rule* (*FMR*), is a resolution strategy that uses a priority to determine the rule action. If a packet matches two rules and the resolution strategy is FMR, the action of the rule with highest priority is executed.
- E is the set of external data associated to the rules. These data are used by the resolution strategy to determine the action, in case of multiple packet matches (e.g. priority values, etc.).
- d is the *default action* of the policy. This action is executed when a packet do not match any of the rules in the policy.

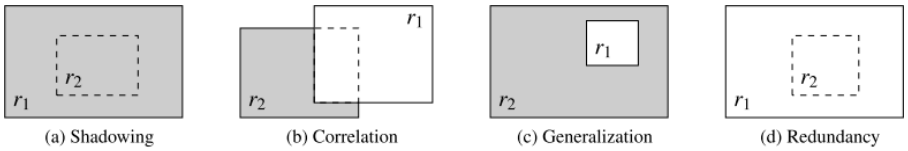

Figure 2.6. Policy anomalies with geometrical rule representation

Figure 2.6 shows how the policy anomalies can be represented using the geometrical rule representation. Modelling policy rules geometrically, enables union, intersection, set minus operations among policy rules. These action are used to find rule relations and anomalies, implementing *resolution* and *reconciliation* algorithms on the policies.

Semantics-preserving morphism

The concept of *semantics-preserving morphism* or simply *morphism* is a transformation of a the policy representation that keeps the policy unchanged [16].

The objective of the morphism is to find an equivalent simplier and optimized policy representation, that can become useful if a policy contains a large number of rules. The resulting policy uses FMR as resolution strategy, so the concept of *rule precedence* is still present with this approach.

To achieve this result, the first step is to generate from the original policy an intermediate representation called *canonical form*. This representation is based entirely on set operation, which helps in conflict resolution and policy manipulation.

A *composition* between rules is generated when two rules overlaps, it is made by the condition clause intersection of the two rules and the action resulting by the application of the resolution strategy. Given a policy, is possible to define its *closure* as the set of all the possible composition of the policy rules.

Using the *semi-lattice* representation of a policy, the management of the canonical form of a policy is easier and more intuitive [16].

The result of this manipulation is a new policy composed by a set o rules ordered by priority. The first rule has the minimum priority value that correspond to the higher priority.

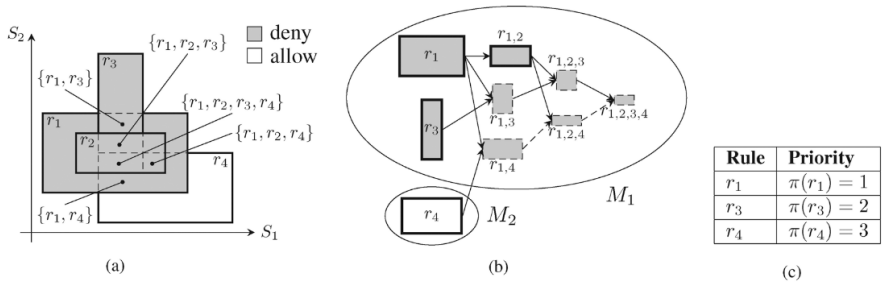

Figure 2.7. A policy FMR-morphism example

As shown is Figure 2.7, the morphism starts from the original policy expressed in its canonical form *(a)* with the corresponding rule action. Then with the semi-lattice representation a *cover-graph* is generated *(b)*. Finally, the resulting rule are exported with the respective priorities *(c)*. It's clearly visible that the rule r_2 is removed from the original policy due to its redundancy with the rule r_1.

With morphism is possible to reduce the number of rules in a policy, removing irrelevant and redundant rules.

Policy reconciliation

Policy *reconciliation* is the process that takes as input a set of policies and produces in output a single resultant policy. This process can be extremely useful if there are multiple filtering points across a single flow path because it produces a single equivalent policy.

Reconciliation also resolve contradictions between rules in different policies. The resulting policy contains the set of *non-conflicting* rules and for each resolved contradiction it generates a new rule.

Reconciliation is done using the semi-lattice representation, the same used for FMR-Morphism technique because it simplify the policy representation and allow to manage easily precedence constraints between rules.

This approach needs complete access to all the policies in order to manage them and produce a single resulting policy. If some policies cannot be accessed or modified is necessary to employ the *policy chaining*.

The process consist, firstly, in a creation of a new rule set containing all the rules of the policies. Then it is necessary to define a *reconciled resolution strategy* because the resolution strategy used in the policies only works on rule from the original policy rule set.

After the reconciled resolution strategy is defined, the resulting policy will contain the set of non-conflicting rule and the set of rules produced by the resolution strategy. Therefore, the last step is to define a default action, using the reconciliation strategy between the default action of the input policy set.

Reconciliation is extremely useful if it is necessary to compute the resulting policy of a series of filtering point. It also exposes all the inter-policy anomalies and correlations that could remain hidden before the process.

Implementing a preventive policy analysis can be extremely useful for network security administrators, especially if there are a large number of policies that need to be enforced. These mechanisms expose anomalies, conflict and allow administrators to solve them getting lighter policies with a reduced number of rules.

Moreover, finding intra-policy or inter-policy conflicts and anomalies is extremely important in computer security, because if a conflict remains hidden, it may lead to serious vulnerabilities and possible external attacks.

Chapter 3

State of the art

In this chapter there is an overview of the *state-of-the-art* of the modern network virtualization technologies.

3.1 Network Function Virtualization

Today's networks are filled with a massive and ever-growing variety of network functions that coupled with proprietary devices, which leads to network ossification and difficulty in network management and service provision. Network Function Virtualization (NFV) is a promising paradigm to change such situation by decoupling network functions from the underlying dedicated hardware and realizing them in the form of software, which are referred to as Virtual Network Functions (VNFs) [1].

This paradigm forces Telecommunications Service Providers (TSPs) to re-invent their networks, building it more dynamic and service-aware.

NFV allows for the consolidation of many network equipment types onto high volume servers, switches and storage, which could be located in data centers, distributed network nodes and at end user premises [2].

A service can be divided into different VNFs, implemented in software, running in various physical servers. This functions can also be relocated from a data center into another, without the necessity to buy specific hardware.

NFV is based on three main pillars.

- Decoupling software from hardware: dedicated hardware is no longer required, because VNFs are implemented in software running on Commercial-Off-The-Shelf (COTS) hardware. Using this approach is possible to develop independently hardware components and network softwares. TSPs can use virtualization techniques to optimize their resources according with their needs. They can also use an hybrid approach mixing virtualized resources with physical ones.

- Flexible deployment: without the necessity to have hardware and software on the same device, is possible to assign the same physical machine to different functions at time. It is also possible to run the same software on different machines with the same result. Services deployment become faster and flexible, with better performances and short configuration times.

- Dynamic scaling: this separations allow services to scale better, especially for virtualized enviroments, because is possible to assign quickly more or less resources, depending on the actual necessity. TSPs can offer services according to traffic conditions, or geographical localization of the customers (e.g. deploying services in a specific data center).

NFV can bring benefits to network carriers, reducing capital investment and energy consumption, implementing a lower-cost agile network infrastructure. It can reduce the *time to market* of a new service, changing the classic deployment cycle and introducing services based on customer necessities.[3]

One of the main challenges of NFV paradigm is how to migrate from the existing and consolidated network infrastructure to virtualized enviroments ensuring the same performances and making the customer unaware of the change.

3.1.1 ETSI NFV Architecture

In October 2013 the European Telecommunications Standards Institute (ETSI) released a document that explains the NFV architecture. This document shows that NFV can be divided into, NFV Infrastructure (NFVI), NFV Management and Orchestration (MANO) and Virtual Network Function (VNF). In the following Figure 3.1, the ETSI NFV reference architecture, is represented.

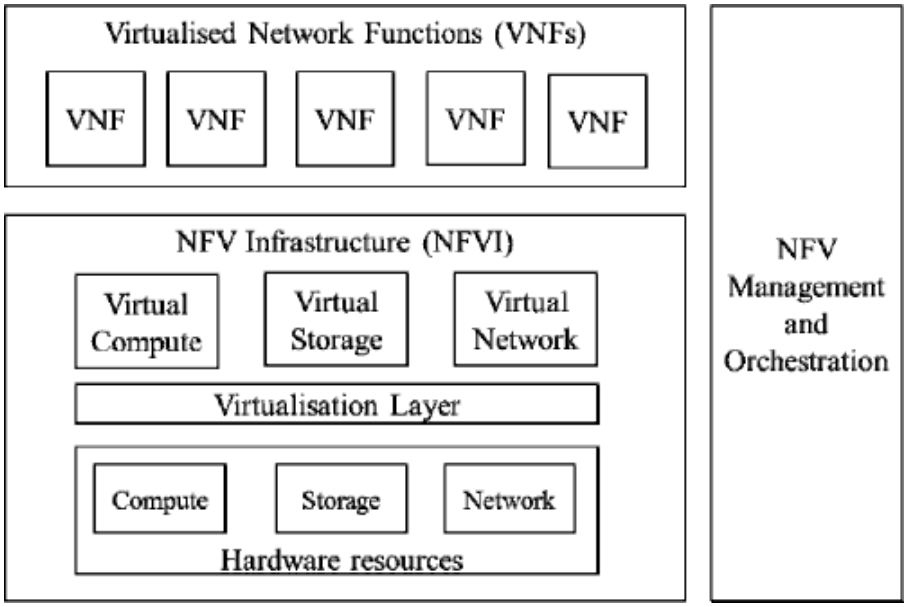

Figure 3.1. ETSI NFV reference architecture [4].

- NFV Infrastructure (NFVI): corresponds to data plane, it forwards data and provides the physical resources for running VNFs. It is composed of COTS hardware which provides the compute and storage capacities and network elements to guarantee the communication between the nodes. There is a virtualization layer between physical infrastructure and virtual resources. It exploits hypervisors to split physical resources across virtual machines. This layer guarantee a level of isolation between VMs sharing the same underlying hardware. Virtual resource layer is divided in virtual storage, compute and network. VNFs runs on these virtual resources in various VMs.

- NFV Management and Orchestration (MANO): is responsible of coordination of the entire NFV architecture. It handles the virtualization mechanism, life cycle of VNFs instances, distribution of hardware resources, modules and interfaces connections, etc. It is divided in Virtualized Infrastucture Manager (VIM) that controls physical resources, NFV Orchestrator (NFVO) that manages VNFs life cycle and assign NFVI resources and VNF Manager (VNFM) that manages the instances of VNFs. This roles are partially overlapping, so they are usually implemented in a single entity.

- VNF Layer: it contains all the VNF instances, it abstract the underlying physical layer.

These instances are organized by Element Managers (EMs). EMs realizes monitoring, configuring functions for a specific VNF. It's collaboration with VNFM is crucial for correct and efficient VNFs deployment.

3.1.2 Virtual Network Function

A VNF is the virtualization of a Network Function (NF), it has to provide the same functionalities of the physical implementation of it. Exploiting cloud computing paradigm is possible to divide VNFs in components, like microservices, and run them on several VMs across a single or multiple datacenters.

According to NFV paradigm, VNFs have to follow some technical requirements to work properly. These can be summarized in four main categories.

- Performance: VNFs usually runs on general-purpose servers, therefore performances can be different from the corresponding physical version implemented on the hardware device. It may be necessary to build efficient algorithms to split network services across different VNFs in various VMs. To guarantee certain levels of throughput or latency, is necessary a continuous communication between VNFM and underlying NFV infrastructure,responsible to gather real-time network performance informations.

- Manageability: VNFs should be instantiated in every moment and in every location by the NFV infrastructure, it should also dynamically allocate and scale physical resources for them. VNFs can be easily interconnected with simple interfaces achieving Service Function Chaining (SFC) between them. To ensure a good level of service quality, network operators may provide redundant instances: this requirement can be achieved using recent Cloud Computing techniques. Also running software should be re-designed to bring crucial informations about performances to NFV infrastructure.

- Reliability and Stability: moving into virtualized enviroment could bring several reliability problems. Network operators should guarantee the same Service Level Agreement (SLA) with customers, implementing automatic handlers for all the new points of failures generated by this migration to virtual appliances. Software must be implemented assuring resilience and stability. Migration is another important parameter for VNFs, in fact all instances should be migrated safely without data loss or service unavailability.

- Security: as VNFs are implemented on third-party data centers, data protection became a crucial aspect for network operators. Furthermore the physical layer is shared between VMs, bringing important isolations problems. This increase the overall load on Intrusion Detection Systems (IDSs). New security threats can be introduced by the underlying network and storage hardware. Is important to ensure that all the instances are isolated from each other since a failure/attack in one service does not affects other instances. Moreover, these software-based components may be offered by different vendors, potentially creating security holes due to integration complexity [3].

3.2 Software Defined Networking

Modern networks are composed by a set of network devices, designed in hardware with Application Specific Integrated Circuits (ASICs) and chips required to achieve high throughput and performances. However, these devices presents lack of flexibility and extensibility. A network operator is forced to use a set of predefined commands to configure these devices and it is not possible to support different protocol or applications.

With hardware devices is difficult to implement policies like *traffic-shaping* or *pay-per-use* services, furthermore, ensuring the continuity of service in case of maintenance operations, is an hard challenge.

To overcome such limitations, a new idea of "programmable-newtorks" is emerging, in particular the concept of Software Defined Networking (SDN). SDN can be defined as: "an emerging architecture that is dynamic, manageable, cost-effective, and adaptable. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services" [5].

Open Network Fundation (ONF) defined a reference model for SDN architecture, it consist in three layers: infrastructure layer, control layer and application layer, as illustrated in Figure 3.2.

- Infrastructure layeris composed of smart switching devices compatible with upper layers and protocols. These devices are responsible of the data plane, they perform packet processing according to the rules given by the SDN Controller, they also provide informations about network topology, network usage and real-time traffic statistics. Unlike common network devices, these devices can perform a programmable *per-flow forwarding* (e.g. they can forward packets matching certain conditions to a specific output port).
- in the control layer are placed the SDN Controllers. It is a layer that bridges the communications between Application layer and Infrastructure layer. It provides Application Programming Interfaces (APIs) to the upper layer. Through this APIs, applications can gather information about network usage and use them to configure controllers. Controllers access to Infrastructure layer devices via protocols that allows the remote configuration of them. This layer is responsible of east-bound communication with control planes that not are not running SDN to maintain compatibility with classic networks.
- Application layer contains SDN user applications. With these application is possible to use the north-bound interfaces brought by Control layer to configure underlying switching devices. These applications can implement standard network functions (e.g. firewall, router, load balancers, etc.). With SDN application is possible to implement also policies of *traffic shaping* configuring a different *per-user* packet processing.

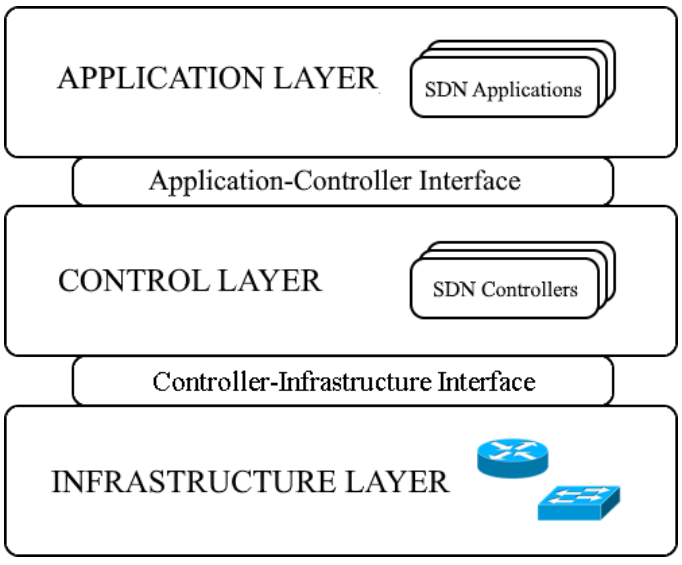

Figure 3.2. SDN Reference model

It is possible to create multiple virtual SDN networks (vSDNs) on a single physical network. The network is split into multiple "slice" and each of them is assigned to a tenant. Each tenant can operate with his "slice" independently from other tenants.

Controllers can be centralized or distributed. In case of distributed controllers is necessary to define the *controller-to-controller* communication. A single, centralized controller represent a single point of failure, thus it may be useful to define backups controller in case of failures.

SDN, with the separation of control plane from data plane, brings several benefits to today's network management activities. It can enhance configurations, allowing for automatic and dynamic network management. It can implement packet forwarding not only at *switching* level, but also at data link layer, breaking the barrier of layering [7]. Performances can be incremented with SDN networks thanks to real-time network status and centralized control. Network operators have a "big-switch" vision of the underlying network, thus they can implement optimized and adaptive algorithms for traffic steering, increasing the overall performances. Lastly, SDN offers a platform for new network designs experimentation due to its capability to create and manage isolated virtual networks.

SDN and NFV are complementary technologies because the former ensures smart traffic steering and the latter provides virtualizable and scalable network functions. These paradigms have similar goals, but they use different approach to achieve them, NFV focuses more on computing while SDN focuses on optimized network path.

3.2.1 OpenFlow

OpenFlow is the protocol that defines the interfaces between Infrastructure layer and the Control layer. It ensures the communication between the SDN Controller and the underlying OpenFlow switch with a secure TCP channel.

OpenFlow switches contains one or more *flow-tables* whose flows can be inserted, deleted or updated by the controller via OpenFlow. This flows consist of *match-fields* used to match incoming packets, this fields can be headers, ingress port or metadata. For each flow is present a set of *actions* that must be applied on each matching packets, these action can modify packet fields, drop the packet or simply forward it to another port. *Counters* are also present in OpenFlow flows, they collect network statistics that can be used to increase the overall SDN network performance.

Rule injection can be done proactively (before receiving packets rules are generated by the controller and loaded into the switch) or reactively (when the switch receive a packet it sends it to the controller that will generate the proper rule).

OpenFlow supports *fine-grained* or *aggregated* rules, it depends on what granularity is needed and where is located the OpenFlow switch (e.g. backbone, edge).

When a packet doesn't match a flow it can be forwarded to another table/rule, sent to the controller or dropped, this behavior depends on the instruction written in the *table-miss*.

With OpenFlow it is possible to define multiple controller for a single switch and a single controller can manage different switches.

3.3 Network Security Function

According to the Internet Engineering Task Force (IETF) definition: "Network Security Function (NSF) is a function that ensures Confidentiality, Integrity and Availability (CIA) to network communications" [8]. Its goal is to block or mitigate malicious activities.

NSFs can be provided to the customers by different vendors or they can be open source technologies, moreover these functions can be implemented on physical or virtual infrastructures. To achieve automatic adaptation, distribution and deployment of NSFs, NFV technologies can be heavily used.

These functions can operate at every level of OSI Protocol stack. With NSFs is possible to provide services as firewalls, Intrusion Prevention/Detection Systems (IDS/IPS), Application Visibility and Control (AVC), Sandboxes, Distributed Denial of Service (DDoS) mitigation, etc.

Multiple NSFs can be combined together to create a security service provided to the customer.

NSFs should provide a set of Security capabilities that are independent from the security controller that will manage them. This capabilities should be expressed in a *vendor-neutral* way, so is not needed to know products and technologies during network security design.

Service providers need standard interfaces to manage and orchestrate NFSs. Interface to Network Security Function (I2NSF) aims to realize these required interfaces.

3.3.1 Interface to Network Security Functions (I2NSF)

In 2014 the IETF started a working group to define a standard model for a set of software interfaces that allows service providers to manage NFSs. According to IETF definition [8]: "The goal of I2NSF is to define a set of software interfaces and data models for controlling and monitoring aspects of physical and virtual NSFs, enabling clients to specify rulesets."

To manage and control NSFs is necessary that the controller is capable to inject rules or execute query according to the requirements.

I2NSF divides the interfaces in two levels:

- Capability layer, indicates how the controller can manage and monitor NSFs. NFSs have to support this rules defined from I2NSF;
- Service layer, indicates how clients can send security policies to the controller, that will apply them according to his capabilities defined in Capability layer.

A client can interact with the controller, via interfaces defined in Service layer, or directly with the NSFs using the Capability layer interfaces.

Chapter 4

Goals

In this chapter the thesis objectives are analyzed, introducing an application of the project on a real use-case.

4.1 Goals definition

Network Function Virtualization (NFV) and Software Defined Networking (SDN), as shown in Section 3.1 and Section 3.2 are fast-growing technologies that are taking part in the modern network and infrastructures design processes. Moreover, with the rapid growth of cloud-based applications, the amount of traffic handled by datacenters will increase as well.

These technologies are constantly improved and updated to ensure high performances under heavy traffic loads. The development of new infrastructures and protocols (e.g. OpenFlow) ensures short response times, fast traffic routing and reliable connections. The same does not apply to security-related operations. Network administrators, in most of the cases, are forced to configure manually a large-number of devices, which is known to be an error-prone task. Moreover the security systems do not provide the same flexibility as the network operations.

Figure 4.1 shows a reference scheme of a traditional security architecture.

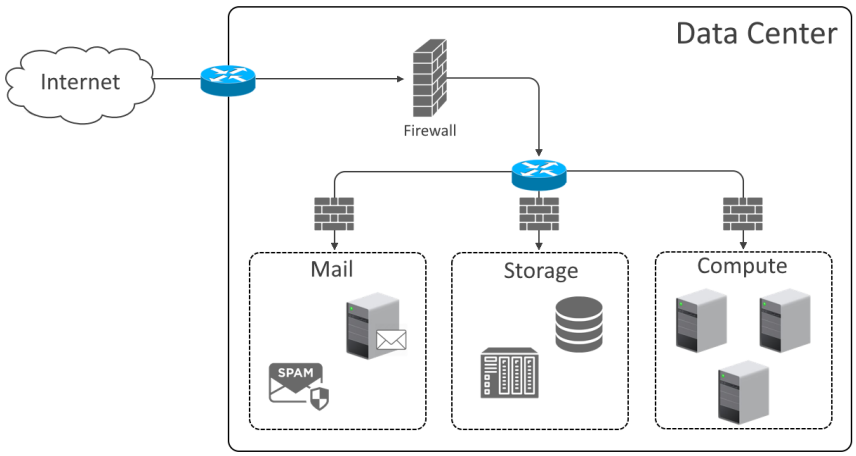

Figure 4.1. A reference scheme for a traditional security architecture

Security requirements are often defined by the user using high level policies, but the distribution process is not as flexible as the policy definition. There are some proprietary systems that perform filtering policy distribution over cloud infrastructures, but they are often not optimized to scale up with high numbers of rules.

This thesis work is focused on the realization of a security system that performs an optimized distribution of high level policies on the provided cloud infrastructure. The objective is to realize a security controller, compatible with most of the cloud architectures, that optimizes the rule distribution process.

By means of this tool an administrator will not have to manually configure the security devices, reducing the possibility of misconfigurations. Moreover, if the topology of the infrastructure changes, the tool can reconfigure all the firewalls automatically enforcing the same security policy.

The SECurity-as-a-Service (SECaaS) paradigm is the key point of this thesis work, because the idea is to implement a service that is agnostic to the technologies used in the cloud environments. Given the model of the infrastructure (physical hosts, virtual switches, virtual machines, etc.) and the high level security policies, the tool will perform an optimized rule distribution.

This project embraces the *microservices* paradigm, splitting a centralized security module into small independent services that communicates each other.

One main idea to improve over the state of the art, is to exploit the filtering points already present in a cloud environment to split the security policies into multiple devices. With this approach, it is possible to avoid increasing the number of devices in the infrastructure, reducing the overall costs.

As an additional requirement, this thesis aims at making the tool as close as possible to a product, compatible with every cloud architecture, easy and intuitive to use. The project will bring an overall simplification of the security operations, without decreasing the strength of the protection.

The final objective is to simplify all the security operations, removing the necessity to manually load the filtering rules on each device. Policies can be defined by the customer remotely, and once they are received by the security controller it distributes them automatically over the infrastructure. The process could lead to significant performance improvements, short response times and an easy security management.

4.2 Use-case definition

In this section a definition of the possible use-cases related to this project is provided. The use-cases are a possible representation of the utilization of this project in a real business environment.

Nowadays small and big enterprises are migrating to cloud infrastructures to host their business services. In this case the amount of employed virtual resources may be huge, so an adequate security system is needed. Moreover, a single enterprise can be divided into several semi-independent entities to manage different business functions.

According to the enterprise internal architectures, two main use-cases were identified: Standalone and Holding company.

Standalone company

This section shows how the optimization tool could improve the security performances in a small company environment.

A small company usually has all the business sectors under a single hierarchical structure. This model can be reflected to the cloud infrastructure used by the company.

In a standalone company the regulations and standards are valid for the entire structure. In the same way the security policies need to be applied to all the resources. In this case concentrating all the security policies in a single filtering point, should be avoided for several reasons.

- *Single point of failure*: if the element that applies the security functions goes down (e.g. attacks, maintenance, etc.), the entire security can be compromised.
- *Performances*: the number of rules may become relevant and aggregating them in a single point may reduces the overall performance, requiring more powerful and expensive resources.

In this scenario, an optimized distribution process splits the rules into multiple filtering points, lightening the single device and gaining in performance.

Distributing the security policies over the infrastructure contributes to reduce the possibility that a cyber-attack can compromise the entire structure. In case of attack it is possible to perform a different distribution, isolating the targeted area and continuing the business activity. Moreover, it is possible to apply the *Defense-in-depth* paradigm, creating multiple layers of defensive mechanisms improving the overall protection.

All the modification to the rule distribution affects the performances, so limiting the overall number of rules is a crucial task that needs to be handled to guarantee a certain performance level.

Holding company

The second use-case concerns the application of the optimization tool in a big, structured business company.

An holding company is a company that is composed by several semi-autonomous *business units*. In this case all the sub-sectors of the holding company are independent with autonomous regulations and governance policies. The main enterprise, however, issues a series of regulations that the business-units have to follow.

This infrastructure can be reflected into a cloud environment, where the main company rents from a Cloud Service Provider (CSP) a series of services. Then the holding company may assign to each unit a portion of the cloud infrastructure on which apply their policies.

In this case there are multiple security policies that needs to be loaded into the cloud environment. Starting from the CSP policies, they have to be applied to the entire datacenter, the holding policies have to be applied to all the underlying business-units, finally each autonomous entity has its personal security requirements.

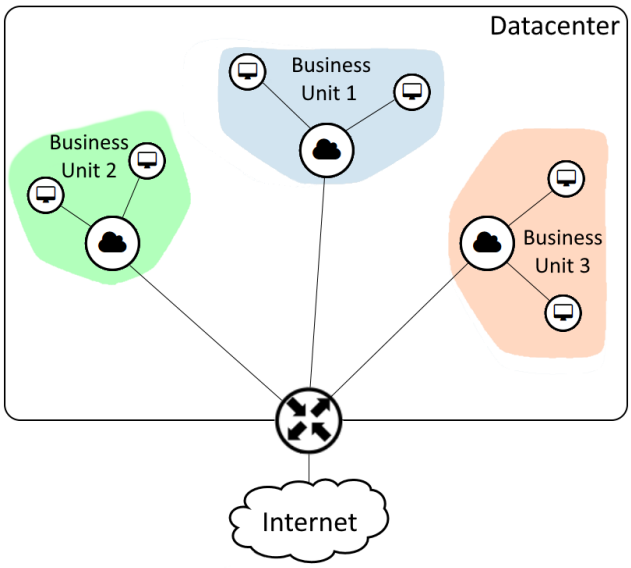

Figure 4.2. A reference scheme for a holding company cloud architecture

The optimization process removes all the conflicts that could be present between the different policies. The distribution process loads the rules only in the filtering points interested by the traffic pattern identified by the rule. This leads to a performance improvement, because the number of rules that needs to be evaluated by the single filtering device is reduced.

As illustrated in Figure 4.2, if an holding company is composed by three business-units, and for each unit a private network is assigned (i.e. `10.0.0.0/24`, `20.0.0.0/24` and `30.0.0.0/24`). The cloud provider filtering rules could be distributed on the "most-external" device, because it have to filter all the traffic directed to the datacenter. The holding company security policies could be distributed on the filtering points that connect the private subnets to the datacenter because all the business-units have to follow main company policies. Finally, the filtering rules related to a specific entity could be loaded only to the devices connected to the respective subnet (e.g. 10.0.0.0/24 related rules should not be loaded into `20.0.0.0/24` devices).

Chapter 5

Components analysis

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms [17].

It provides an intuitive web dashboard, providing to the administrator an easy control and management of the infrastructure.

It is composed by different plug and play components that can be activated when needed. They provide different services and they are all controllable from the same dashboard or command line interface (CLI). Moreover OpenStack provides a set of REST APIs to control and manage all the plugins.

In the thesis project, a basic deployment of the OpenStack suite was used, with compute, deployment, network, security and storage services.

- *Compute*: for computing, the Nova plugin, using `kvm` as virtualization method;
- *Network*: the network management service is Neutron, using `openvswitch` as switch virtualization method;
- *Storage*: for storage, Cinder plugin with a 500GB dedicated hard disk.
- *Image store*: all the images used for instances deployment, are managed by Glance service.

After OpenStack installation, it was analyzed how virtual switches and virtual machine were deployed on the physical hosts.

5.0.1 Computing

Virtual machines are deployed on the `compute` nodes, it is up to the Nova service choose what is the best machine where to launch an instance.

The deployment of new instances can be easily done via web dashboard with which it is possible to assign the required resources (e.g. CPU, RAM, Storage memory) and to select the image to boot from the Glance store.

When an instance is booted up is possible to get SSH access via the dashboard or via CLI from the network node. Moreover it is possible to register some SSH keys to the instance to get secure access from different hosts.

5.0.2 Networking

With Neutron plugin is possible to create *internal* or *external* networks. The former represent a virtual network used inside OpenStack, to ensure communication between virtual machines, usually not reachable from outside. The latter is a *network slice* of the external, usually public, network. A router interface it usually attached to the external network, this allow the virtual instances to communicate with external destinations.

All the virtual machines are connected to a virtual switch called `br-int`. This switch, is usually an OpenFlow switch, but OpenStack supports other type of devices (e.g. Linuxbridge). This virtual device performs all the traffic forwarding operations and can be considered, for each physical OpenStack node, the central networking point.

The network node is the physical host responsible for network management and virtual machines communication. It implements all the virtual routers used to interconnect virtual subnets. This particular node, has an extra virtual switch, other than `br-int`, called `br-ex`. This switch it is used to connect all the OpenStack infrastructure to the external public network.

It is possible to associate a "floating IP" address to an instance, making it reachable from external sources. A "floating IP" is an IP address of the external network addresses pool. After the IP assignment the instance will have two IP addresses: the one that belongs to the internal subnet and the one that belongs to the external network.

The communication between OpenStack physical hosts is done by a VxLAN tunnel. Another virtual switch, called `br-tun`, is responsible of tunnel management.

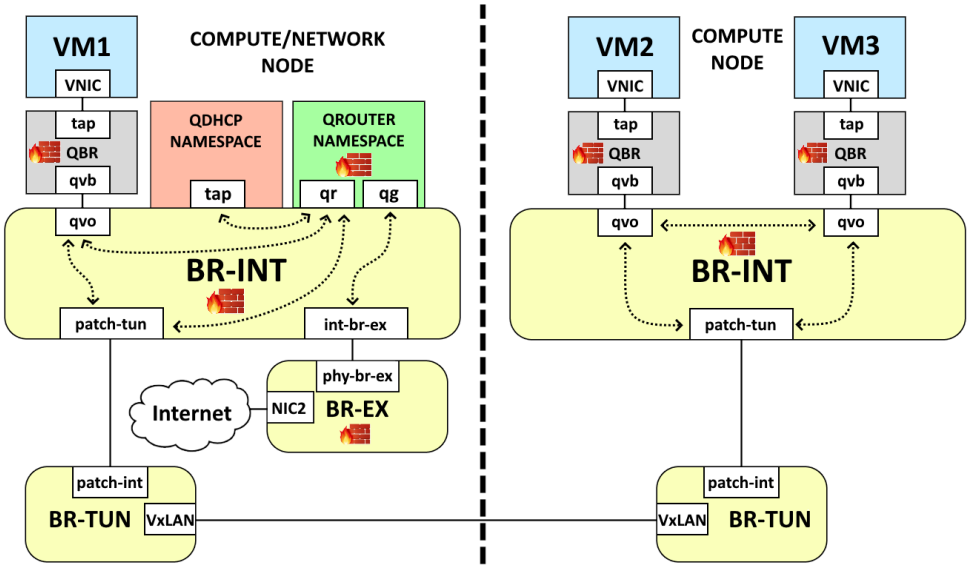

Figure 5.1. OpenStack internal architecture

Figure 5.1 can be used as a reference model to understand OpenStack internal architecture and packet flows.

Packet flows

First of all, is necessary to understand how the communication is realized between physical hosts. A packet that needs to travel from different OpenStack hosts, goes through the `br-tun` switch where it is enveloped into the tunnel packet. Then, the packet is sent, through the primary network interfacem, to the destination host. When the packet is received, it is extracted from the tunnel envelope and it is processed according to the destination.

To better understand what filtering points are available to perform security operation, different communication scenarios were identified. This cases summarize the possible communications between OpenStack instances and Internet.

- Service-to-Service: this traffic pattern represent the internal communication between two services inside the same virtual machine. In this case the packet remains inside the instance. Therefore, from outside, the packet is completely hidden.
- VM-to-VM: this traffic pattern represent the internal communication between two OpenStack instances. This case produces different behavior depending on which physical host the instances are running, and which subnets they belong to.
  - Same subnet: if the two instances belong to the subnet, the traffic path depends from which physical host is running the virtual machines.
    - Same host: if the two instances are on the same host, the packets go through the `br-int` switch directly to the destination instance. Using Figure 5.1 as example, a packet from *VM2* to *VM3*, goes through the `br-int` switch via `qvo` virtual interfaces, reaching the destination directly.
    - Different hosts: to reach an instance on a different host, is necessary to pass through the `br-tun` switch using the VxLAN tunnel. For example, if *VM2* sends a packet to *VM1*, the packet travels through `br-int` switch, it reaches `br-tun` switch via `path-tun` interface and from there the destination host with the modalities explained above. When the packet is received, it is sent to the `br-int` switch via `patch-int` interface and it finally reaches the destination instance.
  - Different subnet: if the two instances belong to different subnet is necessary to pass through the Network node. When the packet reaches this node, it is sent to the router namespace that performs the routing operations and send back the packet to the destination address. For example if *VM2* and *VM3* belong to different subnets, when the packet reaches the Network node, it is forwarded to the *qrouter namespace* via the `qr` interface and the is sent back, after routing operations, to the destination.
- VM-to-External: this traffic pattern represent the communication between an instance and the external network. In this case, the packet reaches the network node as previously explained and from there it is sent to the `br-ex` switch via the `int-br-ex` interface. Through the secondary network interface, the packet is sent on the external network and from there to the "external world" as a normal packet. Moreover, if the instance has a "floating IP" associated, the source address of the exiting packet will the associated IP address. Otherwise, if the instance is internal, the source IP address will be the IP address of the virtual router interface exposed on the external network.

Understanding traffic flows is fundamental to design a security controller that can optimize the rule distribution across the filtering points of the infrastructure.

5.0.3 Security

OpenStack implements a security mechanism based on security groups. A security group is a set of filtering rules, if it is added to an instance during the creation process, the instance will perform security actions according to the rules in the security group.

The implementation of this security system is done by OpenStack using Iptables. Iptables is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel. Several different tables may be defined. Each table contains a number of built-in chains and may also contain user-defined chains. Each chain is a list of rules which can match a set of packets. Each rule specifies what to do with a packet that matches [21]. The rule examination process is iterative, so if a rule do not match the following rule is examined.

In particular, for each virtual instance two Iptables chain are created: one to handle *ingress* traffic and one to handle *egress* traffic. If a security group is assigned to a virtual machine all the

rules loaded into the security group are converted in Iptables rule and loaded in the respective chain.

All the traffic from the standard `FORWARD` chain all the traffic is redirected to the `neutron-openvswi-sg-chain` chain and from there to the respective ingress or egress instance chain.

Figure 5.2 represents the default security group management view of the web dashboard. From the management page is possible to add/remove rules, it is also possible to create new custom security groups and assign them to the instances.

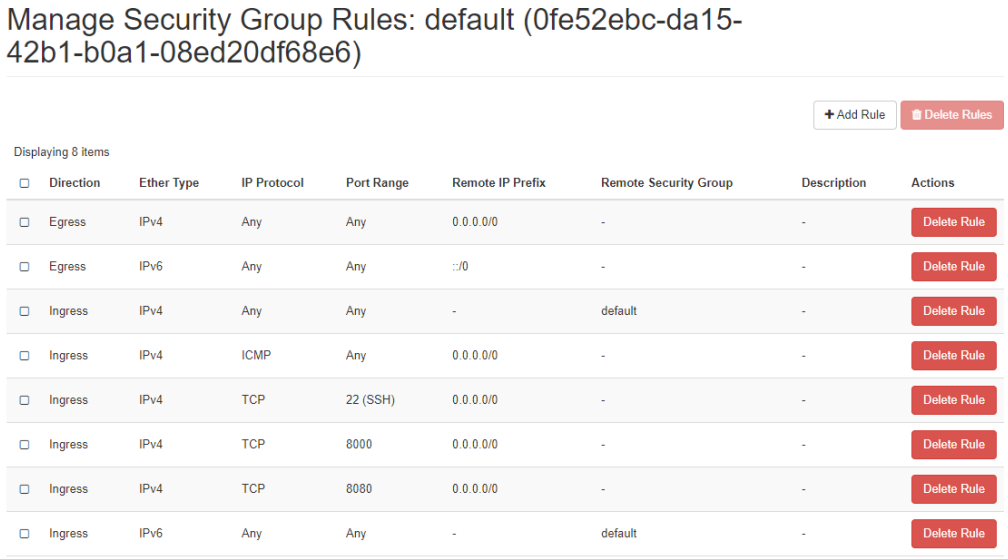

Figure 5.2. OpenStack security groups management page

In the following example is shown how the security group is represented in Iptables for the instance `20.0.0.144`. In particular the *default* security group allows ICMP, SSH, Web Ingress connection and all the Egress connections.

```
$ Chain neutron-openvswi-i102f5ce6-b (1 references)
$ target prot opt source destination
$ RETURN all -- anywhere anywhere state RELATED,ESTABLISHED
$ RETURN udp -- anywhere 20.0.0.144 udp spt:bootps dpt:bootpc
$ RETURN udp -- anywhere 255.255.255.255 udp spt:bootps dpt:bootpc
$ RETURN tcp -- anywhere anywhere tcp dpt:http-alt
$ RETURN tcp -- anywhere anywhere tcp dpt:ssh
$ RETURN icmp -- anywhere anywhere
$ RETURN all -- anywhere anywhere match-set NIPv40fe52ebc-da15-42b1-b0a1- src
$ RETURN tcp -- anywhere anywhere tcp dpt:8000
$ DROP all -- anywhere anywhere state INVALID
$ neutron-openvswi-sg-fallback all -- anywhere anywhere
```

In this case the default rule is `DENY`, so all the unmatched traffic will flow into another Iptables chain (`neutron-openvswi-sg-fallback`) that has only one action:

```
Chain neutron-openvswi-sg-fallback (8 references)
target prot opt source destination
DROP all -- anywhere anywhere /* Default drop rule for unmatched traffic. */
```

Filtering points

From the OpenStack infrastructure analysis multiple filtering points were found on which distribute security policies. As shown in Figure 5.1 is possible to perform filtering actions in various location with different technologies.

In particular, it is possible to apply filtering actions inside an instance (Iptables), in the virtual switches (OpenFlow) and in the physical hosts (Iptables) of the OpenStack deployment.

This aspect can exploited to increase performances, because a single filtering point can be splitted in multiple ones according to the filtering rule matching fields.

The problem with Iptables is its low efficiency when the number of filtering rules grows. Performing some performance tests and using the network bandwidth as metric, it emerged that it decrease as a negative exponential with the number of rules.

OpenVSwitch (OVS) technology uses a cache based data-structure to store the flows. In particular, in OVS an *exact-match cache (EMC)* stores the action performed on a packet, avoiding to search to the corresponding flow when another packet with the same header enters the switch. Moreover all the flows a stored in an hash data structure that uses the packet header as hash key (Datapath classifier), to speed up the searching process. In Figure 5.3, a representation of the internal OpenVSwith architecture is shown.

Iptables performs a linear search, resulting slower than OVS in the searching process, on the other side rule injection in Iptables is faster because it does not need to fill a complex data structure like OVS.

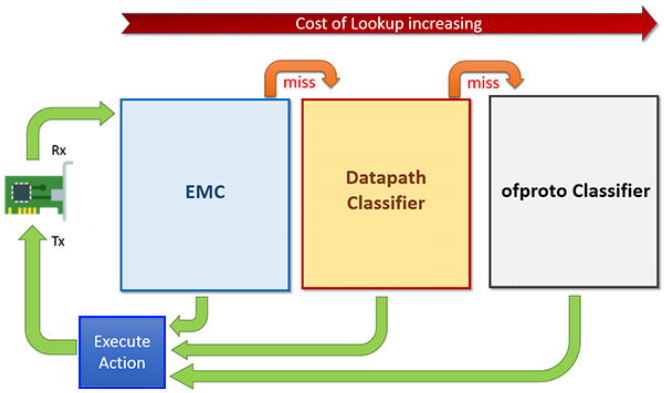

Figure 5.3. OpenVSwitch internal data-structures

The analysis of the filtering points available in a standard OpenStack deployment and the various technologies used, brought to the project several starting points to build an optimization model and consequentely the distribution process. In particular the guidelines to design the security controller focus on the OVS optimization process and fast lookup, limiting the Iptables functionalities to the strictly necessary.

Chapter 6

Solution design

This chapter explains the design of the project and the implementation choices.

6.1 High level design

With the infrastructure analysis, several filtering points emerged. They can be used to optimize the rule distribution across the infrastructure, avoiding to concentrate all the filtering policies on a small amount of physical nodes.

In a cloud environment, as mentioned in Section 4.2, security policies comes from various entities (e.g. CSP, enterprise administration or autonomous business units). It is therefore necessary to processing the policies, removing unnecessary rules or policy conflicts. A policy conflict may weakens the overall security, making some elements vulnerable, so they need to be addressed. Moreover, performing operation between the policies, reduces the total number of rules that needs to be injected. This leads to a performance gain in injection time, optimization time and resource consumption.

In Figure 6.1, a reference scheme for the project design is represented.

Policies are managed by the external library *PolicyToolLib*. Using this library it is possible to handle all the intra/inter-policy conflicts, removing them or notifying the user if an anomaly is not addressable.

Before implementing the distribution process is necessary to understand how and where the rules should be injected.

For this purpose, in collaboration with another student, a distribution model and an optimization model was realized. The aim of the optimization model is to compute the the better place where rules needs to be injected, to optimize certain performance parameters.

The model took as input the physical infrastructure and the security policies and creates two models.

- *Distribution model*: in this phase, a distribution model is created. This model contains, for each rule, the list of available filtering point to inject the rule. A filtering point is elected as available if is on the traffic path related to the filtering rule. If a filtering point is not on the traffic path, it will not receive a single packet of the communication, so is not considered for the distribution process.
- *Optimization model*: in this phase, the optimization model uses the data in the distribution model created, to optimize rule distribution. The outcome of this operation consists in a set of rules for each filtering point that need to be injected. It optimize the rule injection according to the filtering point performances data and to the distribution model.

After this operation, for each filtering node, a list of rules to inject will be available, so the distribution process can start.

The distribution process consists in identifying the type of the filtering point on which the rules need to be injected. It uses two modules that handles the rule loading using Iptables or OpenFlow. According to the technology used in the filtering point, the process performs the conversion of the filtering rule into the set of necessary commands or messages.

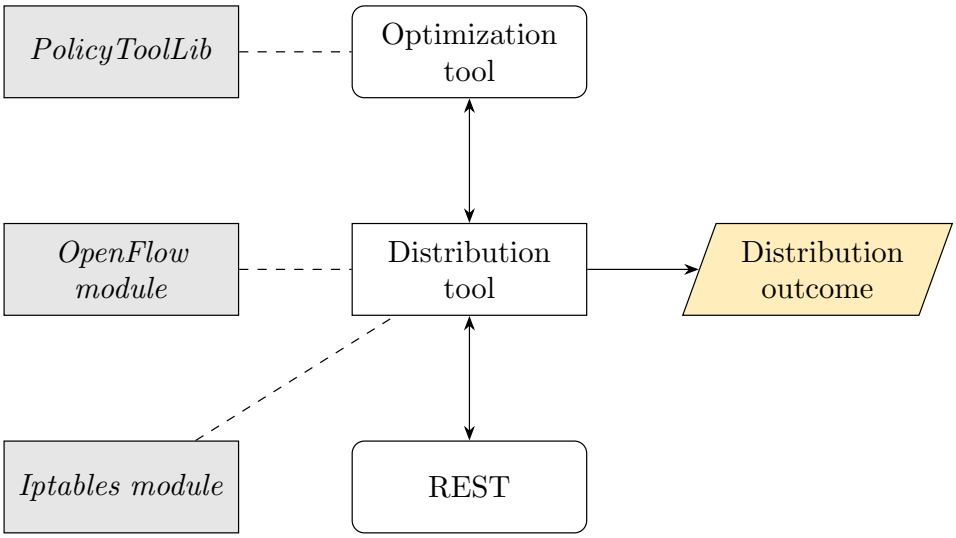

Figure 6.1. High level design scheme

The tool provides a REST service that allows the users to manage the data-strcutures used in the distribution process. Moreover, it give information about the distribution process and offers some API to manage the life cycle of the tool.

In this thesis work, the distribution process is designed and implemented. At the end of the implementation a series of performance tests are done to see if the distribution process brought improvements to the overall infrastructure performances.

6.2 Distribution design

The optimization process produces a set of rules for each filtering point in the OpenStack deployment. Each filtering point category needs a different distribution process design because it uses different technologies.

For Iptables rules we need to distinguish from instances and physical hosts, because despite the final commands are the same, it is not always possible to reach the virtual machine via SSH connection.

In the case of virtual switches the technologies used are completely different, so it is necessary to implement an OpenFlow controller to manage flows injection to the switches.

In both cases is necessary a translation from the filtering policy to the commands to execute to inject the rules on the target host or virtual switch.

6.2.1 Iptables

Iptables is a tool, present in Linux kernel, it implements a series of traffic filtering functions using a simple set of commands.

According to Iptables *man-page* [21], there some useful commands that allow the user to customize filtering rules and tables in the host:

- `iptables -L [chain]`: list all rules in the selected chain. If no chain is selected, all chains are listed;
- `iptables -F [chain]`: flush the selected chain (all the chains in the table if none is given;
- `iptables -N chain`: create a new user-defined chain by the given name;
- `iptables -X chain`: delete the optional user-defined chain specified;
- `iptables -A chain rule-specification`: append one or more rules to the end of the selected chain;
- `iptables -D chain rule-specification/rulenum`: delete one or more rules from the selected chain;
- `iptables -I chain rulenum rule-specification`: insert one or more rules in the selected chain as the given rule number.

Therefore, it is necessary to perform a translation from a series of filtering rules to the list of necessary commands. A validity check is necessary to build correct commands without producing errors or malformed rules.

These commands needs to be executed directly on the host that will perform filtering actions. In this work, Iptables rule injection was divided in two scenarios, according to the distribution/optimization process outcome.

Physical hosts

If a rule needs to be loaded into one of the OpenStack deployment hosts, we need to divide this scenario in two more cases. These cases depends on which hosts the rule will be loaded compared to the host on which the tool is executed.

- *Local host*: if the filtering rules will be injected on the same physical host on which the tool is running, the Iptables commands can be launched locally;
- *Remote host*: if the target host is a remote host, it necessary to open an SSH connection to that host and launch Iptables commands remotely.

For this reason two different rule injection mechanisms are implemented, the former based on a local command execution and the latter based on a SSH connection.

Loading filtering rules on the physical hosts should be avoided because the deployment nodes have to handle the virtualization mechanisms other than all the OpenStack infrastructure. Moreover they have to handle all the traffic that flows in the infrastructure, so the performances are not as good as the virtual instances.

Instances

To inject rules on virtual machines is necessary to get SSH access to them. In cloud architectures, like OpenStack, it can be a problem, because an instance can be launched on a private virtual network, unreachable from the external networks.

OpenStack offers a web-console, allowing administrators to control virtual machines directly from the dashboard. This approach is extremely useful for a quick, intuitive and simple instance management. For this project, instead, this type of console access is not applicable to an automatic optimization tool, so it was necessary to find another way to get access to virtual machines console.

Exploring OpenStack internal architecture, it is possible to understand that the Network node is responsible of the connection between the instances and the "external world". In particular every virtual router in the deployment (if its routes are correctly configured) can reach every instance in the deployment, altough they are in private networks.

Every virtual router is represented as a Linux *network namespace*. On namespaces, it is possible to launch commands in the same way as the standard CLI. This mechanism was exploited to get SSH access to the virtual machines from every host that runs the program. These commands can be only launched on the Network node because is the only node of the OpenStack deployment that has the access to the virtual routers of the topology.

This procedure is divided in two sub-cases depending on which host the tool is running:

- *Network node*: if the tool is running on the Network node the commands to get access to instances can be launched locally;
- *Other nodes*: if the tool is not running on the Network node these commands need to be executed using an SSH connection.

Is important to notice that if a virtual machine has a "floating IP" exposed on the external network, rules can be injected directly on the instance. The procedure is the same previously described as "Remote host" rule injection.

Requirements

To guarantee security to SSH connection we imposed two constraints:

- *Administrator account*: every physical host or virtual machine must have an account that is enabled to launch Iptables commands, on which log in to from SSH connections;
- *SSH Keys*: to guarantee secure connections all the instances and the physical hosts must have the SSH key of the host on which the tool is running, saved as "authorized-keys". In this way, password authentication is not required, improving connections security.

6.2.2 OpenFlow

OpenFlow switches were born to perform fast, context-based forwarding. The forwarding decision is taken based on the flows loaded into the device. This flows consist of *match-fields* used to match incoming packets, this fields can be headers, ingress port or metadata. For each flow is present a set of *actions* that must be applied on each matching packets, these action can modify packet fields, drop the packet or simply forward it to another port.

Flows are grouped in tables, these tables are ordered by the *table-id* parameter. When a packet is received it explores the tables until a matching flow is found and the corresponding action is taken. If no flows are found the default action is taken.

The flows are saved using a multi-level hash structure, using the match fields as a key the corresponding flow is retrieved from the table without performing a linear lookup. Moreover, if a packet matches a flow, its header is saved in a cache along with the taken action. In this way if another packet with the same header is received, the corresponding action is taken without perform the flow table lookup.

The flows can be loaded into the switch from the provided CLI interface, or by using a remote controller via TCP connection. A virtual switch can be connected to several controllers and the single controller can manage different switch a the same time.

We decided to use these flows as filtering rules using only two actions:

- `DROP`: to drop the matching packet;
- `NORMAL`: to let the standard network stack processing the packet;

Flows can be injected from an OpenFlow controller directly into the virtual switch, the injection process is done with a *flowmod* message that contains the flow parameters that need to be injected. A flowmod message is composed by various parameters:

- *Match*: the fields of the packets used to match the flow
- *Action*: the action to execute on the matching packet
- *Flow properties*: these parameters are used to specify additional properties such as the *priority* of the flow, the *table* on which load the flow or the *timeout* after which the flow is removed from the table.

Managing flows injection it is possible to realize a stateless packet filetring directly inside the virtual switch, gaining in performance due to the efficiency of this devices.

In this work the OpenFlow flow injection was divided in three main phases.

- *Controller initialization*: in this phase the controller is initialized, all the parameters are configured. At the end of this phase the controller will be running, with all the virtual switches connected to it and ready to receive new flows.
- *Rule conversion*: in this phase the security policy rules are converted into OpenFlow compatible messages. A single rule can be represented with multiple flows.
- *Flow injection*: in this phase all the generated messages are sent to the virtual switch. The controller sends message via TCP connection initialized in the startup phase.

Requirements

To work properly it is necessary that all virtual switches in the deployment has configured, as controller, the host that will run the distribution tool. This can be easily done specifying the IP address and the port of the TCP connection on which the controller is going to listen for new connections.

6.3 Workflow

In this section is shown the distribution process workflow, starting from the end of the optimization process, through the distribution outcome.

Starting from the list to rule to inject is necessary to understand what kind of filtering point is the target. In case of an Iptables rule injection the process depends on the location of the target filtering point. If the command can be executed locally, it is done via `Runtime` Java class, otherwise the `Jsch` library is used to create an SSH channel and launch the commands remotely.

To inject rules in OpenFlow virtual switches it is necessary to generate the corresponding messages that will be sent to the devices with the `Floodlight` library. The library provides all the methods to instantiate a complete OpenFlow controller that implements all the necessary functions to manage the virtual switches. For sake of clarity, the controller initialization procedure is omitted from the workflow scheme, shown in Figure 6.3.

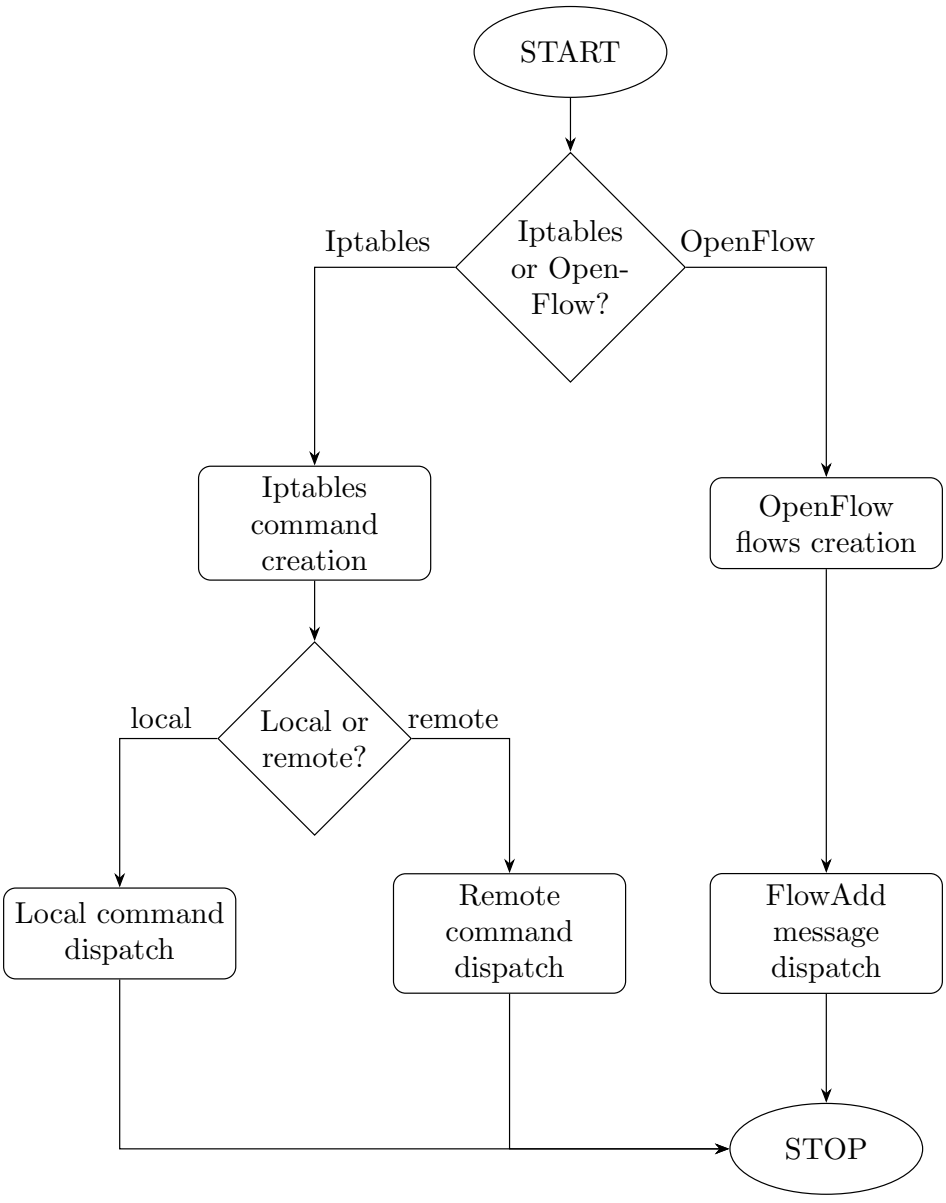

Figure 6.2. Distribution workflow

6.4 REST service

To manage all the tool functions, a REST web-service was implemented. This service allows the users to manage the interaction with the optimization tool, loading input data or getting distribution outcome information.

6.4.1 REST

Definition

REST is acronym for REpresentational State Transfer. It is architectural style for distributed hypermedia systems.

As shown in Figure 6.3, it is usually implemented as a web-server interface on which perform HTTP Requests. It communicates to the main application engine to process the requests and it give back a response according to the operation outcome.

An interface to be considered as RESTful need to follow these principles:

- *Client-server*, separating user interface from the internal data computation, a RESTful service is portable across a large variety of devices;
- *Stateless*, all the requests can not be based on the server contents, and should contain all the informations needed to be processed;
- *Cacheables*, the requests should be marked as cacheable or not-cacheable. If a resource is cacheable the client can use its cached response data for the following requests;
- *Layered system*, all the resources should be disposed in hierarchical layers such that each component can interact only with its immediate layer;

Resources

The key abstraction of information in REST is a resource. Any information can be represented as a resources (e.g. documents, images, etc.). To identify a resource a *resource identifier* is used., while a *media-type* is used to describe the type of the resource.

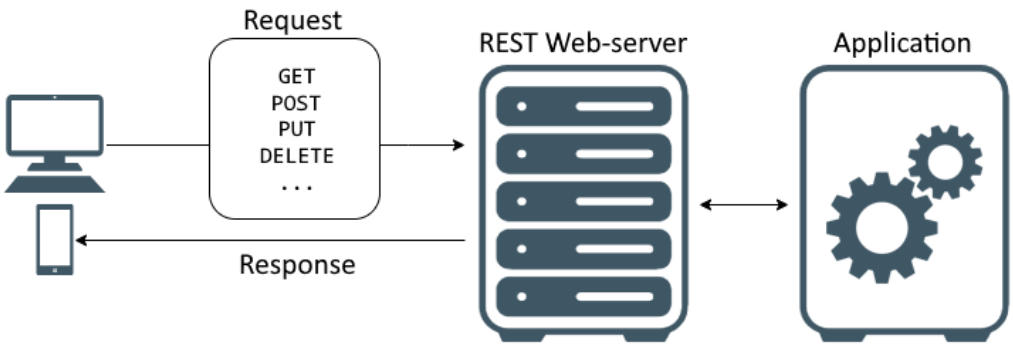

Figure 6.3. REST architecture reference scheme

Usually REST resources are identified by the URI (Uniform Resource Identifier), using them it is possible to reach all the resources directly. It resemble a directory tree reflecting the level of hierarchy.

Further, resource representations shall be self-descriptive: the client does not need to know what represents the resources. It should act on the basis of media-type associated with the resource.

Methods

To get resource informations, update a resource or delete them, it possible to use all the HTTP methods available (e.g. `GET`, `POST`, `DELETE`, etc.). Using them through a request generator (e.g. Postman) all the operation can be performed, and the resources can be edited or visualized.

The Hypermedia as the Engine of Application State (HATEOAS) paradigm is a component of the REST application architecture that distinguishes it from other network application architectures. With HATEOAS, a client interacts with a network application whose application servers provide information dynamically through hypermedia. A REST client needs no knowledge on how to interact with the server except for the understanding of hypermedias.

In this work all the resources provides a set of links that can be used to reach another resource directly, without requiring additional informations.

6.4.2 REST service design

The optimization tool used to getting the list of rules to distribute into multiple filtering points needs several inputs data. These input files (XML files) represents all the information that the tool needs to perform an optimization process.

The inputs can be divided in five main resources:

- `Deployment`, contains all the physical topology resources such as physical hosts, virtual switches and service nodes;
- `Landscape`, contains all the logical topology resources such as firewalls, filtering zones, virtual hosts and links;
- `Entities`, contains all the entities that are managing the infrastructure, for each entity there is a security policy containing different rules;
- `FilteringPoints`, contains the list of exclusions from rule distribution and the list of service node enabled to L7 filtering (middleboxes);
- `SelectorTypes`, contains the selector types used for rule parsing and optimization.

Based on this division, the REST service was designed to bring several methods covering the most important resources needed by the Optimization tool. Moreover an extra section (`Tool`) was added to manage the tool functionalities.

Starting from `deployment` sub-structure is possible to perform `POST` and `GET` requests to load and visualize the entire physical topology. `GET` requests are available for all levels of the deployment structure, but is only possible to execute `DELETE` and `PUT` actions for service nodes. This choice was made because in architectures like OpenStack, to modify physical hosts involved in the deployment is necessary to re-deploy the entire OpenStack suite. Moreover creating new virtual switch and attach them to the infrastructure is not an easy task, so it probably means that the entire deployment will change. Service nodes are prone to continuous changes, for example, in OpenStack, creating or deleting a virtual machine is an easy task that can be executed often and for that reason needs to be supported by REST methods

For the logical deployment (`landscape`) it is possible perform `GET` request to get the entire logical topology, the single service node and the related services. `POST` request are enabled for the entire `landscape`, while `PUT` request can be done only for service nodes and services. The same behavior is applying for `DELETE` request. This choice was taken because adding or remove firewalls, links or filtering zones is not a frequent operation, while updating service nodes by adding/removing services is a common task that have to be supported by the tool.

`Entities` element can be retrieved and created via `GET` and `POST` request, like previous sub-root elements. In this case on all resources is possible to execute `GET`, `PUT` and `DELETE` actions. Therefore an user can create/update/remove an entity with all its rules or perform the same actions on a single rule in an entity security policy.

With the `Tool` structure is possible to manage the program functions like rule distribution, printing a summary of the distribution process or shutdown the services. In particular tool lifecycle is managed via `OPTIONS` requests, while the outcome of the distribution can be retrieved via `GET` request.

The complete list of available REST API can be found in the Appendix B.

Chapter 7

Implementation

This chapter explains how the solution is implemented, from the distribution process to the implementation of the REST service.

7.1 Proof-of-Concept

For the Proof-of-Concept realization, two physical machines were employed. These machines were used to host the OpenStack deployment, dividing equally the modules across them. In particular, the first machine is delegated to the `control`, `network` and `storage` modules. The `compute` and `deployment` modules are running on both machines to divide the amount of resources needed by the instance deployment.

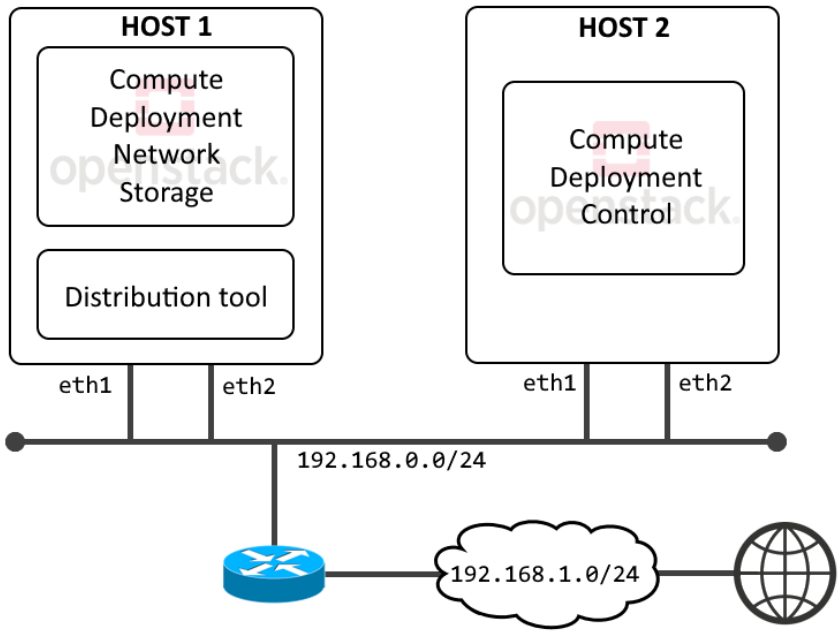

Figure 7.1. Deployed infrastructure scheme

As shown is Figure 7.1 , the machines communicates through a private network (`192.168.0.0/24`). This network is realized through a router connected to the *home-private-network* provided by the ISP.

The distribution tool is running on the network node, because is the most strategic point to reach all the virtual instances of the deployment. Nevertheless, it can be launched on every physical node of the OpenStack deployment, with a little performance reduction due to additional SSH connections.

The machines are two `Intel NUC NUC5i5MYHE`.

- *CPU*: Intel Core i5-5300U, Dual-core, 2.30 GHz;
- *RAM*: 16 GB, DDR3;
- *Operating system*: Ubuntu Server 18.04 LTS;

7.2 Policy representation

Security policies are characterized by a set of filtering rule and a default action that needs to be executed on the packet if it do not match any rules. To perform optimization and distribution operations on the policy it is necessary to find a clear representation of them.

In this work an XML representation is chosen because it better highlight the nesting properties and it is ideal to enhance the element unique attributes. Moreover, XML provides a validation mechanism using an XSD schema, this operation ensure the correctness of the input data, avoid small errors that, in a security context may lead to dangerous situations.

The security policies are divided according to the entity that owns the security policy. These entities can be either the Cloud Service Provider (CSP) or the autonomous companies that are using the cloud infrastructure.

In the following code snippet there is an example of a typical set of security policies.

```
<Entities xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="xml_policy.xsd">

    <Entity Label="ISP" ISP="true" Subnet="0.0.0.0/0">
        <Policy>
            <PolicyName>fw0</PolicyName>
            <PolicyType>FILTERING</PolicyType>
            <DefAction>DENY</DefAction>

            <Rule Action="ALLOW" Label="e0r1">
                <Priority>1</Priority>
                <Selector Label="Source Port">0-65535</Selector>
                <Selector Label="Destination Port">0-65535</Selector>
            </Rule>

            <Rule Action="ALLOW" Label="e0r2">
                <Priority>1</Priority>
                <Selector Label="Protocol Type">1</Selector>
            </Rule>
        </Policy>
    </Entity>

    <Entity Label="entity-1" ISP="false" Subnet="10.0.0.0/24">
        <Policy>
            <PolicyName>fw0</PolicyName>
            <PolicyType>FILTERING</PolicyType>
            <DefAction>DENY</DefAction>
```

```
            <Rule Action="ALLOW" Label="e1r1">
                <Priority>1</Priority>
                <Selector Label="Source Address">10.0.0.0-10.0.0.255</Selector>
            </Rule>

            <Rule Action="ALLOW" Label="e1r2">
                <Priority>1</Priority>
                <Selector Label="Destination
                    Address">10.0.0.0-10.0.0.255</Selector>
            </Rule>
        </Policy>
    </Entity>
</Entities>
```

In the example above there are two entities, the CSP and the *entity-1*. Both security policies are filtering policies and they both have `DENY` as default action, so they will drop all the traffic which not match the `ALLOW` rules.

In particular the provider first rule (`e0r1`) allows all the TCP/UDP traffic with any source/destination port and with any source/destination IP address, while the second rule (`e0r2`) enables the ICMP protocol from any source/destination.

The *entity-1* enables all the IP traffic from and to his subnet (`10.0.0.0/24`).

The result of the optimization process will produce a set of rules that allow TCP, UDP and ICMP protocol limited to the `10.0.0.0/24` subnet. In this way the rules will be injected only on the filtering points related to the *entity-1* subnet and not on others. This could avoid the injection of unuseful rules on filtering points that are not reached by that traffic pattern.

7.3 Distribution implementation

According to the filtering point type (Iptables, OpenFlow) the distribution process has some difference due to the different technologies used.

Rule distribution is divided in two main modules:

- *Iptables*, it is responsible of rule commands building and rules injection/removal for virtual machines and physical hosts;
- *OpenFlow*, it is responsible of virtual switch management and flows injection/removal.

The distribution process is started after the optimization model is complete, and for each filtering point a set of rules to inject are available. The abstract class `FilteringPoint` expose the `public abstract void injectRules()` method that can be called to starts the injection process on each filtering point, according to its type.

7.3.1 Iptables

The distribution process for Iptables starts from the conversion of each `GenericRule` object to inject into Iptables commands. This process produces a set of commands that need to be executed on the target host. At this point, according to the target host type, the commands are launched locally, via SSH connection, or through the network node.

All the classes mentioned in this section are located in the source folder named `it.policy-orchestration.iptables`.

Rule conversion

The conversion process is realized in `IptablesCommandGenerator` and `IptablesParametersGenerator` classes. The former manages the creation or the deletion of Iptables chains and the rule insertion/removal, the latter generates the parameters for the created command (e.g. rule matching fields). Both classes are not instantiable, for that reason they have private constructor.

For example the function `public static LinkedHashSet<String> appendRule(GenericRule rule, String table, String chain)` generates the Iptables command that appends a rule in the chain passed as parameter.

```
1  public static LinkedHashSet<String> appendRule(GenericRule rule, String
       table, String chain) {
2
3      // Getting parameters
4      LinkedHashSet<String> parameters =
           IptablesParametersGenerator.generateParameters(rule);
5
6      // Initialize command set
7      LinkedHashSet<String> commands = new LinkedHashSet<String>();
8
9      // For each parameter insert the iptables command
10     for (String p : parameters) {
11         commands.add("sudo iptables -t " + table + " -A " + chain + p);
12     }
13     return commands;
14 }
```

This function calls the `generateParameters` method of `IptablesParametersGenerator` class. The method perform the conversion from a `GenericRule` to the Iptables command (a `LinkedHashSet <String>` object). This conversion is done getting all the matching fields of the rule and for each field one or multiple commands strings are built

The function returns a set of commands because it is possible that a `GenericRule` object contains a series of matching field not representable with a single Iptables command (e.g. different transport protocols specification), so it may be necessary to build multiple commands for the single filtering rule. The commands are stored in a `LinkedHashSet` data structure, to avoid possible duplicates.

The following code snipped is taken from the mehtod `getSourceIPParameters` of `IptablesParametersGenerator`:

```
1  // For each IP Address range in the selector generate the parameter
2  for (Pair<String, String> ranges : ips.getRangesStr()) {
3      if (list.isEmpty()) {
4          srcIPParameters.add(" -m iprange --src-range " + ranges.getKey() +
               "-" + ranges.getValue());
5      } else {
6          // Otherwise merge previous parameters
7          for (String prev : list) {
8              srcIPParameters.add(prev + " -m iprange --src-range " +
                   ranges.getKey() + "-" + ranges.getValue());
9          }
10     }
11 }
```

In particular this portion of code inspects the *Source IP address* field of the rule and adds/merges it to the Iptables command already built (e.g. with*Destination IP address*). If it is the first field that needs to be converted, the string is simply added to the list without merging with previous ones. The Iptables module `iprange` is used to represent IP address ranges in Iptables commands.

Local dispatch

If commands need to be executed locally the dispatching process is handled by `SystemCommandManager` class. This class is an implementation of the abstract class `CommandDispatcher`. Calling the `public void dispatch(LinkedHashSet<String> commands)` method, the commands passed as argument are launched according to the dispatcher type (e.g. Local, SSH).

The following code snippet shows how the commands are launched locally. The method `exec(String command)` of the `Runtime` class is responsible of command execution. Therefore, the error stream of the process is redirected to a buffered reader, so if an error occurs, it can be catched and an Exception can be thrown.

```
1  // Execute commands
2  for (String c : commands) {
3      try {
4      // Create exec process
5      Process process = Runtime.getRuntime().exec(c);
6
7      // Read output
8      BufferedReader br = new BufferedReader(new
           InputStreamReader(process.getErrorStream()));
9
10     ...
11
12     }
13 }
```

Remote dispatch

To dispatch commands remotely it is necessary to setup an SSH connection to the target host. To do that, we used the *JSch* library [23], it provides all the methods to create custom SSH connections and execute remote commands.

The class `JschSessionManager` is responsible to initialize the connection, it extend `CommandDispatcher` abstract class, so it implements the `public void dispatch(LinkedHashSet<String> commands)` method.

Command injection works in three main phases.

- *Session creation*: in this phase the SSH connection is set up, configuring the key path, username, host and connection port. This is done in the `createSession()` method that returns a `Session` object.

```
1 // Add the private key path
2 jsch.addIdentity(privateKeyPath);
3
4 // Initialize a new session
5 session = jsch.getSession(username, domain, 22);
6
7 // Set key authentication
8 session.setConfig("PreferredAuthentications", "publickey");
```

The last method, sets the authentication method via SSH keys, avoiding password authentication.

- *Channel initialization*: in this phase the *exec* channel is opened and configured for the created sesssion. The exec channel execute a command remotely via an SSH connection. This is done in the `createChannel(Session session)` method that returns a `ChannelExec` object.
- *Command execution*: in this phase the commands are launched on the remote host. All the iptables are concatenated in a single string, avoiding to set up multiple SSH connections. This is done in the `runRemoteCommands(LinkedHashSet<String> commands, ChannelExec channel)` method.

```
1  StringBuffer sb = new StringBuffer();
2
3  // Build commands
4  for (String command : commands) {
5      sb.append(command + ";");
6  }
7
8  // Set commands to the channel
9  channel.setCommand(sb.toString());
10
11 // Launch command
12 channel.connect();
13
14 // Read input to wait command execution
15 checkErrorOutput(channel);
```

When the `connect()` method in called, the commands are executed. If some error occurs the method `checkErrorOutput(channel)` will throw an `SshConnectionException`. If an exception is thrown a rollback process begins removing all the added rules and shutting down the tool. This because a fail on a single rule injection may compromise the entire system security.

Instances dispatch

To launch commands on the OpenStack running instances, it is necessary to prepend `"sudo ip netns <router-namespace-id>"` before each Iptables command. This string executes the commands within the router namespace, in this way, all the instances can be reached while they are in private networks.

This commands must be launched on the Network node (locally or via SSH), because it is the only node that contains all the virtual routers in the deployment. On the network node, it is possible to know which are the virtual routers namespaces typing: `sudo ip netns`.

7.3.2 OpenFlow

The distribution process for OpenFlow switches is done via *Floodlight* Java library. This library implements an OpenFlow controller directly in the Java environment. All the classes involved in this session are in the `it.polito.policyorchestration.floodlight` source folder.

Controller initialization

The Floodlight main module is initialized at tool startup. It implements the `Runnable` class that runs the module in a separate thread. The initialization phase is based on the `floodlight.properties` configuration file, it is possible to choose the modules to launch and to customize some parameters (e.g. connection port, etc.).

All the modules can be easily extended with custom methods, implementing the associated interfaces. After controller initialization the `dispatcher` module is created, it will be loaded into every `VirtualSwitch` object, to start flows injection. The dispatcher module is responsible of the creation of the OpenFlow messages used to add/remove flows in the switches.

The Floodlight main module is started with the following function:

```
1 @Override
2 public void run() {
3     try {
4         loader.runModules();
5     } catch (FloodlightModuleException e) {
6         logger.error("Failed to run controller modules", e);
7         System.exit(1);
8     }
9 }
```

The method is an override of the `run()` method of the `Runnable` class, when it is called the execution starts on another thread gaining in parallelism and performances. It starts all the loaded modules enabling virtual switch connection and flow injection.

In the thesis project three modules are principally used.

- *StaticEntryPusher*: is the module responsible for flows injection. It is in charge to send OpenFlow *flowmod*, ensuring that the virtual switch receives the message correctly.
- *MessageHandler*: is the module responsible of sending/receiving OpenFlow messages. In the project this module was extended to implement the shutdown procedure. When called, it removes all the injected flows and send an `OFBarrierRequest` message to all the connected switches.

```
1 for (IOFSwitch sw : switchManager.getActiveSwitches()) {
2     // Create BarrierRequest Message
3     OFBarrierRequest barrier =
          sw.getOFFactory().buildBarrierRequest().setXid(999).build();
4     // Send message
5     while (!sw.write(barrier)) {}
6     // Update waiting response map
7     waitingResponse.put(sw.getId(), true);
8 }
```

This message, in OpenFlow, is a synchronization message, when received by the switch it performs all the previous queued actions, then sends back an `OFBarrierReply` message. After the messages are sent, all the switches are put in the `waitingResponse` data-structure.

When a reply is received the corresponding switch is disconnected, and when all the switches are disconnected the shutdown procedure continues until the end of the program.

- *SwitchListener*: is the module that handles switch connection and state changes. When a switch is connected to the controller the following method is triggered.

```
1 @Override
2 public void switchActivated(DatapathId arg0) {
3     synchronized (dispatcher) {
4         dispatcher.notifyAll();
5         logger.warn("Notifying waiting rule dispatcher");
6     }
7 }
```

This method wakes up all the `FlowDispatcher` object that are waiting for switch connection before send the `OFFlowAdd` message. All the virtual switches are identified with the `DatapathId`, that is a `6 Byte` identification number similar to a MAC Address.

The following code snippets shows how the `FlowDispatcher` checks, with the `OFSwitchManager` Floodlight module, if the switch with the given `DatapathId` is connected. If the switch is not already connected the Floodlight threads goes in *waiting* mode until `SwitchListener` module wakes it up.

```
1 // This additional while loop is made to prevent spurious notifications
2 while ((sw = switchManager.getActiveSwitch(DatapathId.of(id))) == null) {
3     try {
4         logger.warn("Waiting switch " + id + " become active");
5         this.wait(30000);
6         if (!tries) {
7             tries = true;
8         } else {
9             throw new FlowDispatcherException("The virtual switch is not
                  connected to the controller");
10        }
11    ...
12 }
```

If, after 30", the switch is not connected to the controller, there is a problem in some configurations (e.g. wrong datapath id), so an Exception is thrown and the tool needs to be stopped.

Flows injection

In this phase every rules that will be injected into the virtual switch, needs to be converted into the corresponding OpenFlow flow. In particular the process is done into `FlowDispatcher` class that offers these two methods.

- `public void dispatch(String id, GenericRule rule)`: this method is divided in two main blocks: the former is a `synchronized` block that waits for the virtual switch connection, while the latter generates the message and sends using the `StaticEntryPusher` Floodlight module.

In the following code snippet it is possible to see how the flows are injected into the virtual switch.

```
1 // Getting all flows
2 LinkedList<OFFlowAdd> flows = generateFlows(rule, ofFactory);
3 for (OFFlowAdd f : flows) {
4 String flowname = f.getMatch() + " - " + rule.getName();
5
6 // Push the flow into the switch
7 entryPusher.addFlow(flowname, f, sw.getId());
8
9 // Update the list of added flows
10 flowsAdded.add(flowname);
11
12 logger.info("Flow " + f.getMatch() + f.getActions() + " pushed into
      switch " + sw.getId());
13 }
```

Moreover the created flows are saved into a data-structure, that ensures the flow removal operation during the shutdown process.

- `LinkedList<OFFlowAdd> generateFlows(GenericRule rule, OFFactory factory)`: this method perform the conversion from a `GenericRule` object into one or multiple OpenFlow flows. The conversion uses principally the `MatchBuilder` class provided by Floodlight library to generate the matching field of the flow (e.g. source/destination IP address, Ports, etc.).

  Then for each flows it sets the corresponding action, in particular, if no action is set the corresponding action will be `DROP`:

```
1 // Generate actions
2 if (rule.getAction() == FilteringAction.ALLOW) {
3 OFActionOutput normal =
      factory.actions().buildOutput().setPort(OFPort.NORMAL).build();
4 actionList.add(normal);
5 }
```

When the flow action are set up, it is necessary to setup additional flow parameters as timeouts, destination tables, priorities:

```
1 // Generate flows
2 for (Match m : flowMatches) {
3 flows.add(factory.buildFlowAdd().setBufferId(OFBufferId.NO_BUFFER)
4     .setHardTimeout(3600).setIdleTimeout(3600).setPriority(32768)
5     .setMatch(m).setActions(actionList).setTableId(TableId.of(60))
6     .build());
7 }
```

During creation process, several optimization are performed on the rules that will be injected, especially for transport ports and Ip addresses match fields.

- *Port ranges*: a range of ports it is usually expressed as `"startPort-endPort"`. In OpenFlow notation it is not possible to use this representation. To avoid to create a flow for each port, OpenFlow offers a "masked notation" mechanism to represent port ranges.

  For example, according to the official man-page of the `ovs-ofctl` command [24], the representation of the `1000-1999` range can be implemented with a series of `port/mask` tuple: `0x03e8/0xfff8, 0x03f0/0xfff0, 0x0400/0xfe00, 0x0600/0xff00, 0x0700/0xff80, 0x0780/0xffc0, 0x07c0/0xfff0`, avoiding to inject a thousand of flows. This representation resemble to the "IP address/netmask" notation used to represent subnets.

  This process is implemented into the `PortRangeManager` class.

- *IP address ranges*: Similar to previous section, an IP address range can be represented as a series of subnets. It reduces a lot the total number of flows that need to be generated from a single filtering rule. For example to represent all the addresses from `10.0.0.64` to `10.0.0.85`, instead of using all the twenty-one addresses, only three subnets are needed: `10.0.0.64/28, 10.0.0.80/30, 10.0.0.84/31`. This process ensure a great reduction in number of flows and a consequent performance gain.

  The conversion process is implemented into the `IPRangeManager` class.

7.4 REST service

The REST service is the main interface to interact with the tool. When the tool is launched it starts a web-server on which the REST service will run. The web-server is completely embedded inside the program, so it is not necessary to use external tools to expose the REST APIs.

The particularity of the REST services is that they have to be completely stateless, every method invoked by an HTTP request is not aware to current server state. Moreover this service is implemented following the multi-thread and parallelism approach. The REST module runs on a separate thread and for each new method invocation a new thread is launched.

7.4.1 Multi-threading

This multi-thread approach may cause problems when the data structures are shared between REST threads. When some data can be used by multiple threads a series of synchronization problems may arise (e.g. deadlocks, race conditions, etc.).

In this case, a synchronization problem may lead to inconsistent data representation and wrong results. For example if the distribution process is launched and in the same time some data modification requests are performed, the process may lead to unwanted results or simultaneous access errors.

To solve this problem the `PolicyOrchestrationService` was implemented. This class contains only `synchronized` methods. A `synchronized` method in Java ensures that only one thread at time can invoke it. Using this technique all the data-structures used inside these methods are protected from synchronization problem.

There are two classes that need synchronized accesses to their data: `InfrastructureManager` and `LifecycleManager`.

- `InfrastructureManager` contains all the information about the data-structures used by the distribution tool. It modelize the physical topology, the logical topology and all the filtering policies that needs to be loaded into the infrastructure.

- `LifecycleManager`: it is responsible to the program life cycle, it perform the host initialization and it starts the shutdown process.

Without synchronization these classes may use the same objects leading to an unwanted behaviour.

The `PolicyOrchestrationService` is used not only by the REST server, because all the methods needed to perform optimization, distribution, input/output are defined in it.

Topology management

A typical example to better understand how this class works consist in the update operation of the infrastructure. It is represented by the `InfrastructureManager` class and it is one of the most important parts of the distribution tool. It contains all the informations needed by the tool to distribute filtering rules, for that reason it is necessary that the access to the data structured is synchronized between all the operating threads.

The following code snippet shows how a service node creation is managed by the class:

```
1  public synchronized ServiceNode loadNodeElement(String IP, String label,
       String nodeIP, NodeElement node){
2
3      ...
4
5      // Get deployment
6      Deployment deployment = getDeployment();
7
8      // Get PhysicalHost
9      PhysicalHost h = deployment.searchPhysicalHost(IP);
10     if (h == null) {
11         throw new NotExistingPhysicalHostException();
12     }
13
14     // Get virtual switch
15     VirtualSwitch vsw = h.searchVSwitch(label);
16     if (vsw == null) {
17         throw new NotExistingVirtualSwitchException();
18     }
19
20     // Get performance Factory
21     PerformanceToolFactory pf = new
           PerformanceToolFactory(deployment.getPerformanceType());
22
23     // Create service node
24     ServiceNode s = PhysicalTopologyFactory.createNode(deployment, node, pf);
25
26     // Add service node to deployment
27     vsw.addServiceNode(s);
28
29     ...
30
31 }
```

A `ServiceNode` is an object that represent the node on which services are running. In the case of an OpenStack deployment it is the representation of an instance running in the deployment. This function represent the creation of a new virtual machine in the OpenStack deployment, therefore this instance will be deployed on a specific physical node and then attached to a virtual switch according to the already presented infrastructure scheme.

The first part of the method performs a validation on the loaded data structures to see if the `ServiceNode` object can be created in that position. In particular it checks if the physical node and the virtual switch on which the element will be loaded are present in the `Deployment` data structure.

If the validity check is satisfied the `ServiceNode` is created and added to the specified virtual switch, otherwise an Exception is raised and the rollback operation is done.

7.4.2 Lifecycle management

The REST service can manage the life cycle of the program with two "special" APIs. At the `/api/tool/` URI there are three resources that can be exploited to manage tool functionalities.

- `/start`: this API is reachable with an `OPTIONS` request. It starts the entire distribution process from the optimization tool to the rule injection. It is an unique block because all the actions that are done during this process should be referring to the same data-structure, avoiding all the problems previously mentioned.

  In particular this function takes the loaded data-structures (e.g. topology, policies, etc.) and starts the distribution process. It consist, firstly, in a policy refinement process to remove all the rules anomalies that can be found in a security policy (Section 2.3.2). Then the optimization tool produces for each filtering point the corresponding list of filtering rules. Finally the rules are distributed according to the type of the filtering host.

- `/stop`: this API is reachable with an `OPTIONS` request. It begins the shutdown process, removing all the rules loaded into the filtering points. In particular, this procedure flushes all the created Iptables chain and it remove them.

  Then all the OpenFlow flows are removed, ending with the synchronization procedure with the virtual switches. During this process a `OFBarrierRequest` with a particular ID are sent to all the virtual switches. When all replies are received it means that all the switches have completed all the operation and they can be shut disconnected.

- `/output`: this API is reachable with an `GET` request. It shows the output of the distribution process or an `404 - Not found` error if the distribution process is not completed. The output is an XML file containing a list of filtering points with the list of injected rules.

Chapter 8

Testing

The optimization tool, before starting the distribution process, performs two operations on the given security policies. Firstly it optimizes them removing all the conflicts and the redundancy in the policies, then it perform a *reconciliation* process with the security policy given by the CSP. These operations can reduce drastically the number of rule that needs to be injected into the filtering points.

After the policies are ready to be inserted, the optimization tool realized by Igor Ferretti computes all the filtering points interested bt the rule matching traffic. This operation prevent the insertion insert of a filtering rule in a device not "touched" by that traffic pattern.

These scenarios are used to test out the tool, verifying the performance gain obtained by the optimization and distribution process.

8.1 Test design

The tests are implemented to highlight the differences between OpenStack security and the advantages of using the optimization/distribution tool, realized in this thesis work.

Several metrics are used to measure performances, analyzing which of them are the most important in a business scenario. Moreover, the test-cases are divided in two main sections.

- Policy optimization: this test case shows how an huge set of rules can be reduced into a smaller one. This leads to a gain in performances because the resource needed by the filtering points depends on the number of rule that needs to process.
- Distribution optimization: this test case shows how a set of filtering rules can be distributed according to the traffic identified by the policy. Instead of putting all the rules in the same filtering point, distributing them across the infrastructure leads to a performance gain.

All the tests are performed using an infrastructure that reminds to the real use-case. Several business units are deployed in it with autonomous security policies to recreate an environment similar to the final use of the project.

In some test cases the security policies are filled with unnecessary filtering rules to highlight the optimization process, while in other cases they are not optimized on purpose to highlight the distribution process.

Metrics

To test the performances several metrics are used, in particular the chosen metrics are CPU utilization, network latency and bandwidth. The performances are a key point in a cloud environment both for client and for the provider. A performing network connection with high bandwidth and small latency is always appreciated by the customers, but to achieve this it may be necessary for the provider to have powerful hardware due to the resource consumption required by cloud services.

For each metric measurement a different tool is used.

- *CPU utilization*: for this metric we used the standard `top` program available on every Linux distribution. It provides a dynamic real-time view of a running system. It displays a list of currently running processes and the resources consumption summary [18]. Executing `top` program the shell will print the following output:

```
$ top

$ top - 18:53:02 up 0 min, 0 users, load average: 0.52, 0.58, 0.59
$ Tasks: 4 total, 1 running, 3 sleeping, 0 stopped, 0 zombie
$ %Cpu(s): 0.3 us, 0.8 sy, 0.0 ni, 98.8 id, 0.0 wa, 0.1 hi, 0.0 si, 0 st
$ KiB Mem: 16663516 total, 10118300 free, 6315864 used, 229352 buff/cache
$ KiB Swap: 29221116 total, 29221116 free, 0 used. 10213920 avail Mem

$ PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
$ 1 root 20 0 8892 312 272 S 0.0 0.0 0:00.07 init
$ 7 user 20 0 16776 3440 3328 S 0.0 0.0 0:00.08 bash
$ 39 user 20 0 17624 2048 1516 R 0.0 0.0 0:00.00 top
```

- *Latency*: to measure network latency the `ping` and `nping` tools are used. These tools measure the Round Trip Time (RTT) between two hosts during a standard ping exchange. The RTT is the time needed for a signal to be sent plus the time needed by the acknowledgement of that signal to be sent back. Nping can generate network packets for a wide range of protocols, allowing users full control over protocol headers [19]. To evaluate the latency performances the following command are launched:

```
$ sudo ping -c 1000 -i 0.005 -q -s 1000 destination-IP
$ sudo nping -H --no-crypto -c 1000 --send-eth --data-len 1000 --delay 5ms
    --tcp -p 80 destination-IP
```

  The commands launch a series of `1000` ping messages, every `5 ms`. Every packet size is `1000 Bytes`.

- *Bandwidth*: represents the maximum amount of data successfully exchange in a communication path. To measure network bandwidth the `iPerf3` tool is used. It is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols [20].

```
$ iperf3 -s -f K
$ iperf3 -c server-IP -f K
```

  The first command is needed on the destination machine to initialize the iPerf3 service, while the second command is executed on the source machine and it starts the bandwith measurement.

Infrastructure

According to Figure 8.1 and Figure 8.2, the tests are performed on an infrastructure similar to a real-use case. In particular the cloud environment is divided in four semi-autonomous sub-units. A private subnet is assigned to each entity (from `10.0.0.0/24` for the *entity-1* to `40.0.0.0/24` for the *entity-4*).

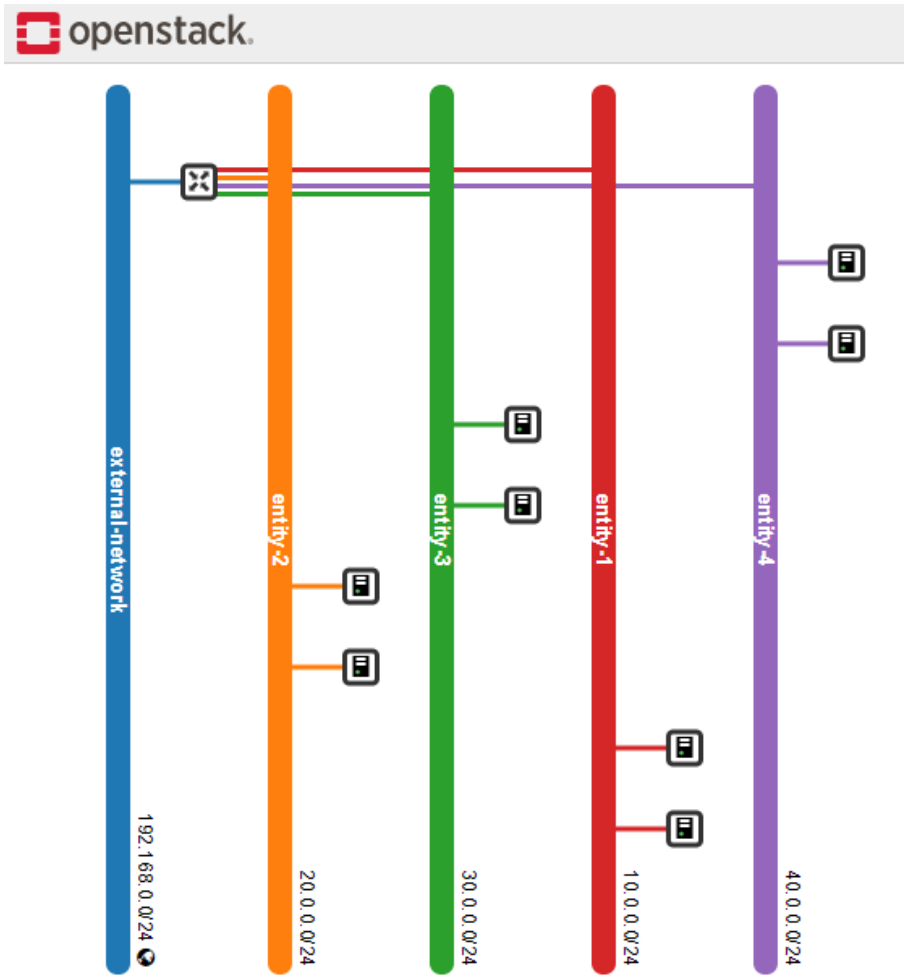

Figure 8.1. The infrastructure used to perform the performance tests

The entities subnets are connected to the external network through a virtual router. This router is responsible to ensure communication between entity subnets and between the VMs and the external destinations. It performs NAT functions, translating the private IP addresses of the VMs into the public address exposed on the external network, making the communication possible.

It is possible to assign to each virtual machine a *floating* IP address, a public address announced by the router to the external network. Using floating IPs it is possible to make the VMs in the private networks reachable from the external networks.

The virtual machines are instances of Cloud Ubuntu 16.04 operating system, each of them is provided with the following specifications.

- *CPU*: 2 Virtual CPUs are assigned to each VM. Using KVM as virtualization type, each virtual CPU consists in a thread assigned to the KVM process that handles the virtualization of the machine.

- *RAM*: Each instance is provided with 1GB of RAM. Due to its optimization for cloud environments Cloud Ubuntu do not requires high resources to perform basic operations and it offers all the functionalities required to perform the performance tests.
- *Storage*: Each VM has a 4GB dedicated virtual disk. The management of the storage disks is managed by the Cinder module using a dedicated 512GB physical hard-disk reserved and formatted ad-hoc.

According to the OpenStack infrastructure reference scheme (Figure 5.1), the instances are deployed on both physical hosts. The location on which an instance will "spawn" is chosen by the Nova scheduler module that computes, when an VM is launched, which is the best host to deploy it. The network node is responsible of the routing operations, it ensures the communication between virtual instances and between the VMs and the public Internet.

Finally, all the instances are interconnected through the *br-int* virtual switches that will be strongly used as a suitable filtering point to apply security policies due to its high performances.

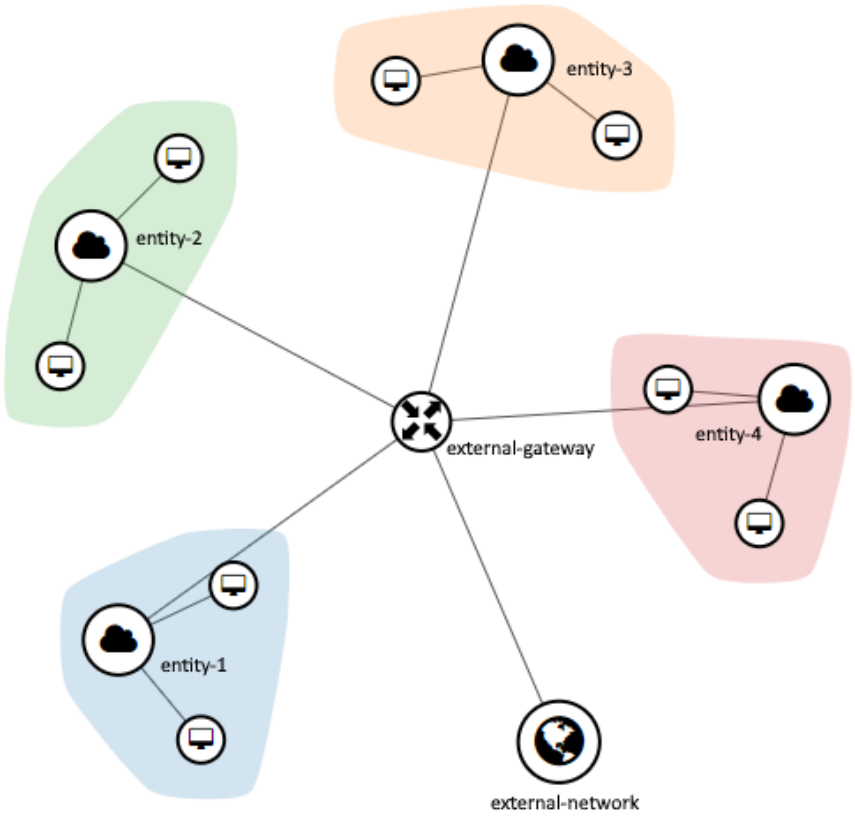

Figure 8.2. Logical representation of the infrastructure

8.2 Policy optimization

In OpenStack it is possible to load security policies using the Security Groups (Section 5.0.3). This operation can be done using the web dashboard, exploiting the provided REST APIs or by CLI. Loading a security policy in OpenStack security group consists in a creation of Iptables rules in the chains related to the virtual machines associated to the security group.

The objective of this section is to show how the policy optimization is a crucial activity that needs to be done before performing the rule distribution. This operation reduces the total number of rule to inject leading to a dramatic increase of the performances. The optimization process is done using the *PolicyToolLib* library.

To demonstrate how the policy optimization works this XML file was used:

```
<Entities xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="xml_policy.xsd">
    <Entity Label="ISP" ISP="true" Subnet="0.0.0.0/0">
        <Policy>
            <PolicyName>fw0</PolicyName>
            <PolicyType>FILTERING</PolicyType>
            <DefAction>DENY</DefAction>
            <Rule Action="ALLOW" Label="e0r1">
                <Priority>1</Priority>
                <Selector Label="Source Port">0-65535</Selector>
                <Selector Label="Destination Port">0-65535</Selector>
            </Rule>
            <Rule Action="ALLOW" Label="e0r2">
                <Priority>1</Priority>
                <Selector Label="Protocol Type">1</Selector>
                </Rule>
        </Policy>
    </Entity>

    <Entity Label="entity-1" ISP="false" Subnet="10.0.0.0/24">
        <Policy>
            <PolicyName>fw0</PolicyName>
            <PolicyType>FILTERING</PolicyType>
            <DefAction>DENY</DefAction>

            <Rule Action="ALLOW" Label="e1rA">
                <Priority>1</Priority>
                <Selector Label="Source Address">10.0.0.0-10.0.0.255</Selector>
            </Rule>
            <Rule Action="ALLOW" Label="e1rB">
                <Priority>1</Priority>
                <Selector Label="Destination
                    Address">10.0.0.0-10.0.0.255</Selector>
            </Rule>
            <Rule Action="ALLOW" Label="e1r1">
                <Priority>2</Priority>
                <Selector Label="Source Port">0</Selector>
                <Selector Label="Protocol Type">6</Selector>
            </Rule>
            <Rule Action="ALLOW" Label="e1r2">
                <Priority>2</Priority>
                <Selector Label="Source Port">1</Selector>
                <Selector Label="Protocol Type">6</Selector>
            </Rule>
```

```
            ...
            <Rule Action="ALLOW" Label="e1r3">
                <Priority>2</Priority>
                <Selector Label="Source Port">10000</Selector>
                <Selector Label="Protocol Type">6</Selector>
            </Rule>
        </Policy>
    </Entity>
</Entities>
```

The XML file defines two entities: the CSP and the *entity-1* associated to the subnet `10.0.0.0/24`. The CSP security policies allows all the TCP, UDP and ICMP traffic while it blocks all the other type of packets, while the *entity-1* one allows all the IP traffic from/to his subnet (rule `e1rA` and `e1rB`), with a series of rules that allows TCP traffic with a determined *Source Port*.

The series of "extra" TCP rules of the *entity-1* are logically unuseful and redundant, because the `e1rA` and `e1rB` allows all the IP traffic, so it is unnecessary to specify accepting rules for transport layer traffic. This choice was made on purpose because it better highlight the differences between OpenStack security mechanisms and the optimization/distribution tool. In this case the performance data are taken starting from `10` filtering rules up to `10000`.

The performances measurements are taken based on the communication between two virtual machine of the *entity-1* in particular the `10.0.0.1` and the `10.0.0.2` instances.

8.2.1 OpenStack security groups

The result of injecting the rules using OpenStack Security Groups is a set of Iptables rules appended to each virtual machine reserved chain. This happens because OpenStack do not perform any optimization on the given security rules, moreover each virtual machine that is assigned to a security group is converted in an Iptables chain on which all the rules loaded in the security group are injected. The chains are spliy in Ingress/Egress, according to the rule parameters.

The following figure shows a portion of an Iptables chain related to the Ingress traffic of a virtual machine running on the deployment. In particular it shows that all the TCP rules are appended to the chain sequentially.

```
...
RETURN tcp -- anywhere anywhere tcp spt:ftp
RETURN tcp -- anywhere anywhere tcp spt:telnet
RETURN tcp -- anywhere anywhere tcp spt:24
RETURN tcp -- anywhere anywhere tcp spt:smtp
RETURN tcp -- anywhere anywhere tcp spt:26
RETURN tcp -- anywhere anywhere tcp spt:27
RETURN tcp -- anywhere anywhere tcp spt:28
...
```

This process leads to an huge amount of rules injected into each Iptables chain, with a consequent performance reduction. This expression shows how much rules will be injected starting from a single security policy:

$$R_i = R_p * C \tag{8.1}$$

where R_i is the total number of injected rule, R_p is the number of rules in the security policy and C is the number of Iptables chains related to the virtual machine assigned to the security group.

8.2.2 Optimization tool

Using the optimization tool the security policies are transformed to remove unnecessary rules, gaining in performances. The first action that is done on the policy is to make the entities policies compatible with the CSP one. This process is done because the CSP security constraints have higher priority against entities, so an entity can not provide security policies that are in contrast with the provider.

In this case the `e1rA` and `e1rB` are merged with the CSP security policies. The result of this operation is a set of rules that enables TCP, UDP and ICMP traffic for the `10.0.0.0/24` network. All the "extra" TCP rules are therefore merged into the TCP enabling rule.

This optimization reduces drastically the number of rules. According to the OpenStack infrastructure, these rules are injected into the virtual machines and in all the virtual switches connected to the *entity-1* dedicated subnet.

```
...
table=60, n_packets=576, n_bytes=123716, tcp,nw_src=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, udp,nw_src=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, icmp,nw_src=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, tcp,nw_dst=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, udp,nw_dst=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, icmp,nw_dst=10.0.0.0/24 actions=NORMAL
...
```

The above picture shows a portion of the flow table of a virtual switch internal to the OpenStack deployment. It is highlighted that the rules are translated to a set of rules that allows TCP, UDP and ICMP traffic from (`nw_src`) and to (`nw_dst`) the `10.0.0.0/24` subnet.

The details of the optimization process can be found in the Igor Ferretti work, that is focusing on the optimization and the distribution model.

8.2.3 Performances

In the following graphs is shown the performance comparison between OpenStack security system and the optimized rule distribution. The tests are done with the previously defined tools measuring the network latency, bandwidth and the CPU consumption.

Latency test

Figure 8.3 shows that the Round Trip Time (RTT), expressed in milliseconds, degrades as a straight line with the increase of the rule numbers.

The Round Trip Time is the amount of time needed by a signal to go to his destination and back. If the number of rules, that need to be processed by the devices on the path through the destination is high, the amount of time needed by the signal to reach his destination increases as well.

From `0` to `1000` rules both OpenStack security and the optimization tool has the same performances. After `1500` rules the two lines diverge drammatically until they reach a difference of `450 ms` at `10000` rules.

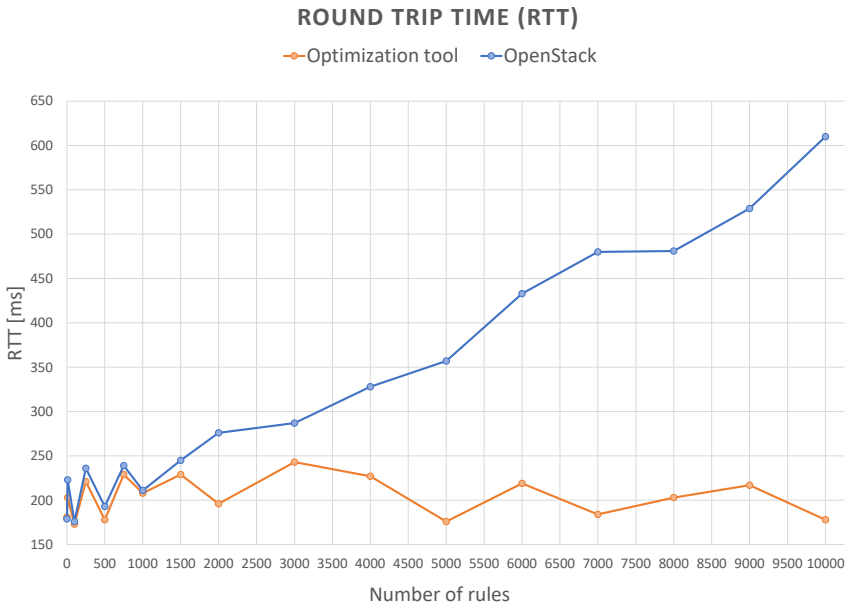

Figure 8.3. Latency test report for policy optimization

This behaviour shows how the high number of rules loaded with OpenStack security groups impact dramatically on the performances. The optimization process consists in the removal of all the redundancies and anomalies between filtering rules in a security policy. For example, if a rule is shadowed by another one, the rule will be removed and therefore not injected in the device.

The test is made to highlight the rule removal process, defining a large set of filtering rules full of anomalies. The results shows that all the additional rules are removed correctly, leaving the policy with only few filtering rules.

Bandwidth test

Figure 8.4 shows that the bandwidth performance degrades as a negative exponential with the increase of the rule numbers. From `0` to `100` rules both OpenStack security and the optimization tool has the same performances.

The network bandwidth can be defined as the maximum amount of data that can be transmitted on a link in a certain amount of time. If the computation time needed by the filtering devices to elaborate the filtering decision is short, the resulting bandwidth is high. This because the in the same amount of time it is possible to process and elaborate more data.

After `100` rules the two lines diverge drammatically until they reach a difference of `1300 MB/s` at `10000` rules.

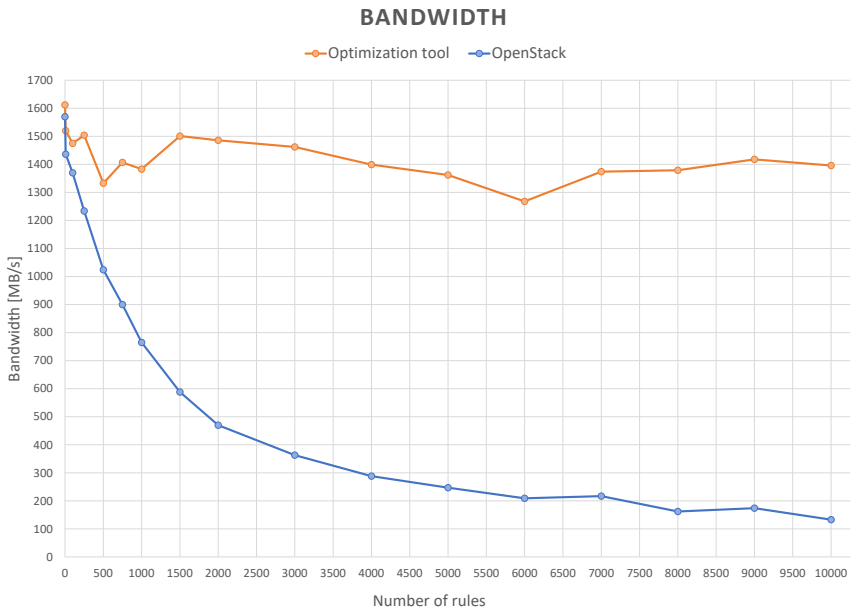

Figure 8.4. Bandwidth test report for policy optimization

When the number of rules increases, the slope can be defined as a negative exponential because of the TCP window scaling. The TCP window is the maximum amount of data that can be transmitted over a TCP connection before getting the acknowledgment back. This window is automatically adjustable by the end-points of the connection, according to the congestion conditions. If the receiver is able to process a large amount of data, the window is extended to its maximum (65,535 B), otherwise the window is shrank to reduce the packet loss.

In this case the TCP window scaling process stabilizes the slope of the network bandwidth, because the amount of packet sent is proportional to the time needed to process them.

As previously mentioned, the optimization process reduces the total number of rules, getting high bandwidth values and maximizing the TCP window.

CPU consumption test

Figure 8.5 shows that the CPU consumption (in percentage) performance degrades linearly with the increase of the rule numbers. Immediately the difference between OpenStack security and the optimization tool became visible.

Injecting only a few rules the CPU consumption remains stable near `0-1%`, while with standard Security Groups the resource utilization relative to the Iptables rule processing grows up to `20%` at `10000` rules.

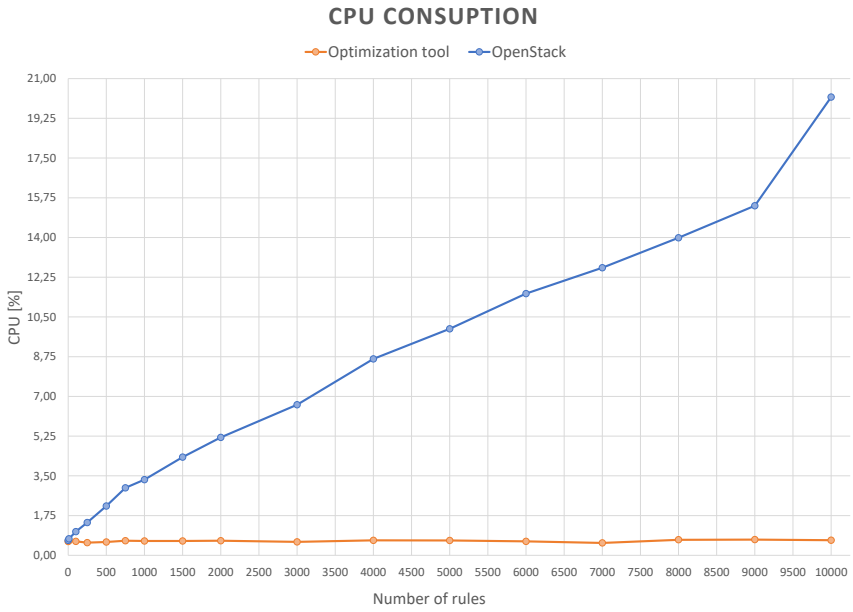

Figure 8.5. CPU consumption test report for policy optimization

The CPU consumption is measured on the device in which the filtering rules are loaded. The value used as a reference for the measurement is the Software Interrupt Context (softirq or si). This value indicates the percentage of CPU used for the processing of the software interrupts, which are triggered by the packet processing actions, like packet filtering.

With the increase of the number of rules, the amount of CPU needed to process the filtering decision became high. To determine the decision, the device needs to analyze all the rules in the security policy, until it finds a matching rule, or until it reaches the default rule.

Optimizing the total number of rules, it is possible to avoid to process unnecessary redundant rules, saving many CPU cycles.

8.3 Distribution optimization

Security policies can contain a list of filtering rules related to different targets. In the use-cases defined in this thesis work a company may be divided in several independent sub units, so for the main company, it can be necessary to define a different security policy for each business unit that owns.

In this section is shown the performance difference resulted in using OpenStack security groups and the optimization/distribution tool. The metrics used for the tests are the same used before, using the same tools. In this test case each entity security policy is filled with a large number of rules to highlight the difference of performances. The objective of this test-case is to show how distributing the rules according to the relative target could bring a performance improvement in terms of Bandwidth and Latency.

Is known that the rules used in this test case can be optimized reducing the total number of them, but this choice was made because the tests focus on the distribution process and not on the policy optimization (as in the previous section).

The XML file used for this test case is the following:

```
<Entities xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="xml_policy.xsd">
    <Entity Label="CSP" ISP="true" Subnet="0.0.0.0/0">
        <Policy>
            <PolicyName>fw0</PolicyName>
            <PolicyType>FILTERING</PolicyType>
            <DefAction>DENY</DefAction>
            <Rule Action="ALLOW" Label="R1">
                <Priority>1</Priority>
                <Selector Label="Source Port">0-65535</Selector>
                <Selector Label="Destination Port">0-65535</Selector>
                <Selector Label="Protocol Type">1</Selector>
            </Rule>
        </Policy>
    </Entity>

    <Entity Label="entity-1" ISP="false" Subnet="10.0.0.0/24">
        <Policy>
            <PolicyName>fw0</PolicyName>
            <PolicyType>FILTERING</PolicyType>
            <DefAction>DENY</DefAction>

            <Rule Action="DENY" Label="e1r1a">
                <Priority>1</Priority>
                <Selector Label="Destination
                    Address">10.0.0.0-10.0.0.255</Selector>
                <Selector Label="Source Address">130.192.0.0</Selector>
            </Rule>

            <Rule Action="DENY" Label="e1r2a">
                <Priority>1</Priority>
                <Selector Label="Destination
                    Address">10.0.0.0-10.0.0.255</Selector>
                <Selector Label="Source Address">130.192.0.1</Selector>
            </Rule>

            ...
```

```
        <Rule Action="DENY" Label="e1r10000a">
            <Priority>1</Priority>
            <Selector Label="Destination
                Address">10.0.0.0-10.0.0.255</Selector>
            <Selector Label="Source Address">130.192.0.100</Selector>
        </Rule>

        <Rule Action="ALLOW" Label="e1rA">
            <Priority>1</Priority>
            <Selector Label="Source Address">10.0.0.0-10.0.0.255</Selector>
        </Rule>

        <Rule Action="ALLOW" Label="e1rB">
            <Priority>1</Priority>
            <Selector Label="Destination
                Address">10.0.0.0-10.0.0.255</Selector>
        </Rule>
    </Policy>
</Entity>

...

<Entity Label="entity-4" ISP="false" Subnet="40.0.0.0/24">
    <Policy>
        <PolicyName>fw0</PolicyName>
        <PolicyType>FILTERING</PolicyType>
        <DefAction>DENY</DefAction>

        <Rule Action="DENY" Label="e4r1a">
            <Priority>1</Priority>
            <Selector Label="Destination
                Address">40.0.0.0-40.0.0.255</Selector>
            <Selector Label="Source Address">130.192.0.0</Selector>
        </Rule>

        <Rule Action="DENY" Label="e4r2a">
            <Priority>1</Priority>
            <Selector Label="Destination
                Address">40.0.0.0-40.0.0.255</Selector>
            <Selector Label="Source Address">130.192.0.1</Selector>
        </Rule>

        ...

        <Rule Action="DENY" Label="e4r10000a">
            <Priority>1</Priority>
            <Selector Label="Destination
                Address">40.0.0.0-40.0.0.255</Selector>
            <Selector Label="Source Address">130.192.0.100</Selector>
        </Rule>

        <Rule Action="ALLOW" Label="e1rA">
            <Priority>1</Priority>
            <Selector Label="Source Address">40.0.0.0-40.0.0.255</Selector>
        </Rule>
```

```
            <Rule Action="ALLOW" Label="e4rB">
                <Priority>1</Priority>
                <Selector Label="Destination
                    Address">40.0.0.0-40.0.0.255</Selector>
            </Rule>
        </Policy>
    </Entity>
</Entities>
```

The XML files defines, other than the Cloud Service Provider (CSP) entity, four other entities (*entity-1*, *entity-2*, *entity-3*, *entity-4*). The CSP, in this test case, is assumed to allow all the TCP, UDP and ICMP traffic, without limitations, while each entity security policy allows the outgoing/ingoing traffic from/to the internal network (rule `e1rA` and `e1rB`).

Moreover each entity blocks the traffic that is coming from a set of IP address, varying on the number of rules chosen for the performance misuration. In this case the performance data are taken starting from `10` filtering rules up to `10000`.

The performances measurements are taken based on the communication between two virtual machine of the *entity-1* in particular the `10.0.0.1` and the `10.0.0.2` instances.

8.3.1 OpenStack security groups

In OpenStack, using security groups, the policies are loaded in the Iptables of the physical hosts that performs the computing operations. In particular for each virtual machine an Iptables chain is assigned, and each rule in the security group is loaded into it.

In this case the traffic is simply forced to flow through the chain and if it matches some rules the corresponding action will be taken.

In the following picture is visible how rules are added by OpenStack to Iptables, in particular is shown the Ingress traffic chain of the `10.0.0.2` instance:

```
target prot opt source destination
...
DROP all -- 130.192.0.4 anywhere destination IP range 10.0.0.0-10.0.0.255
DROP all -- 130.192.0.4 anywhere destination IP range 20.0.0.0-20.0.0.255
DROP all -- 130.192.0.4 anywhere destination IP range 30.0.0.0-30.0.0.255
DROP all -- 130.192.0.4 anywhere destination IP range 40.0.0.0-40.0.0.255
DROP all -- 130.192.0.5 anywhere destination IP range 10.0.0.0-10.0.0.255
DROP all -- 130.192.0.5 anywhere destination IP range 20.0.0.0-20.0.0.255
DROP all -- 130.192.0.5 anywhere destination IP range 30.0.0.0-30.0.0.255
DROP all -- 130.192.0.5 anywhere destination IP range 40.0.0.0-40.0.0.255
...
```

It is clearly visible that for each filtering rule the corresponding Iptables line is generated by OpenStack and loaded into the respective chain. The problem is that the `10.0.0.2` will not receive traffic directed to the `20.0.0.0/24` network, so in this case the number of loaded rules are four time the real need. This causes a performance decrease that could be resolved using an optimized distribution process.

8.3.2 Optimization tool

In this work a distribution optimization process is realized to overcome the previously detected problems. During the analysis several possible filtering points were found to be available to perform security operations.

In particular the distribution process, after the intra/inter-policy optimization phase, takes in account the traffic pattern targeted by the rules and computes an optimal distribution model. The outcome of the process is, for each fitering point, the list of filtering rules that cover the traffic pattern handled by the filtering point. In particular on the Iptables tables of the virtual machines only the rules relative to the instance subnet will be injected.

In the following figures are shown the Iptables of the virtual machines `10.0.0.1` and `20.0.0.1`:

```
target prot opt source destination
...
DROP all -- 130.192.0.4 anywhere destination IP range 10.0.0.0-10.0.0.255
DROP all -- 130.192.0.5 anywhere destination IP range 10.0.0.0-10.0.0.255
DROP all -- 130.192.0.6 anywhere destination IP range 10.0.0.0-10.0.0.255
DROP all -- 130.192.0.7 anywhere destination IP range 10.0.0.0-10.0.0.255
DROP all -- 130.192.0.8 anywhere destination IP range 10.0.0.0-10.0.0.255
...
RETURN icmp -- anywhere anywhere destination IP range 10.0.0.0-10.0.0.255
RETURN tcp -- anywhere anywhere tcp destination IP range 10.0.0.0-10.0.0.255
RETURN udp -- anywhere anywhere udp destination IP range 10.0.0.0-10.0.0.255
DROP all -- anywhere anywhere
```

Figure 8.6. Iptables view of the `10.0.0.1` instance

```
target prot opt source destination
...
DROP all -- 130.192.0.4 anywhere destination IP range 20.0.0.0-20.0.0.255
DROP all -- 130.192.0.5 anywhere destination IP range 20.0.0.0-20.0.0.255
DROP all -- 130.192.0.6 anywhere destination IP range 20.0.0.0-20.0.0.255
DROP all -- 130.192.0.7 anywhere destination IP range 20.0.0.0-20.0.0.255
DROP all -- 130.192.0.8 anywhere destination IP range 20.0.0.0-20.0.0.255
...
RETURN icmp -- anywhere anywhere destination IP range 20.0.0.0-20.0.0.255
RETURN tcp -- anywhere anywhere tcp destination IP range 20.0.0.0-20.0.0.255
RETURN udp -- anywhere anywhere udp destination IP range 20.0.0.0-20.0.0.255
DROP all -- anywhere anywhere
```

Figure 8.7. Iptables view of the `20.0.0.1` instance

In the example, the rules regarding the `20.0.0.0/24` network are not loaded on the `10.0.0.1` instance. Using this approach the traffic flows in less filtering rules than using OpenStack security groups.

The process takes in account also the virtual switches that interconnects the virtual instances as shown in the following figure:

```
...
table=60, n_packets=0, ip,nw_src=130.192.0.4,nw_dst=10.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.5,nw_dst=10.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.6,nw_dst=10.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.7,nw_dst=10.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.8,nw_dst=10.0.0.0/24 actions=drop
...
table=60, n_packets=0, ip,nw_src=130.192.0.4,nw_dst=20.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.5,nw_dst=20.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.6,nw_dst=20.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.7,nw_dst=20.0.0.0/24 actions=drop
table=60, n_packets=0, ip,nw_src=130.192.0.8,nw_dst=20.0.0.0/24 actions=drop
...
table=60, n_packets=0, n_bytes=0, udp,nw_dst=20.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, tcp,nw_dst=20.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, icmp,nw_dst=20.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, tcp,nw_dst=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, icmp,nw_dst=10.0.0.0/24 actions=NORMAL
table=60, n_packets=0, n_bytes=0, udp,nw_dst=10.0.0.0/24 actions=NORMAL
```

Figure 8.8. Virtual switch `br-int` flow table

In Figure 8.8 is shown the content of the flow table of the virtual switch `br-int` that interconnects the instances. Since this switch is the same for all the traffic directed to the virtual machines it is necessary to load all the filtering rules in the policy on the device. However, due to the high optimization done by the OpenVSwitch data structures the flows loaded into the switch are not affecting the performances tests.

The tests are based on a data exchange between two VMs, so the traffic pattern between them is the same for the entire test duration. In this way the packet headers are cached by the virtual switch avoiding to perform for each packet the flow research. Performing a direct access to the resources the performance increase is an immediate consequence.

8.3.3 Performances

In the following graphs is shown the performance comparison between OpenStack security system and the optimized rule distribution. The tests are done with the previously defined tools measuring the network latency and the bandwidth.

Latency test

The following graph shows that, as expected, the performance behaviour is the same for OpenStack and the distribution tool, because both of them inject the filtering rules using Iptables. The difference in performances is noticeable as a different line slope, because the number of rules injected in the single device is lower than using OpenStack.

The distribution process injects the rules into different devices according to the traffic pattern the rules identifies. In this test case the measurements are done on the receiving device, highlighting the differences using OpenStack and the optimization tool. In particular, the target device is on the `20.0.0.0/24` subnet. In this case there are four entities that defines the same security policies. Using the distribution process the amount of rules injected in each filtering point is four times less than using OpenStack.

Under `200` rules the difference is not noticeable, because the computation time is negligible compared to the latency measurement. The difference became relevant after `500` rules, with a difference of `200 ms` at `10000` rules.

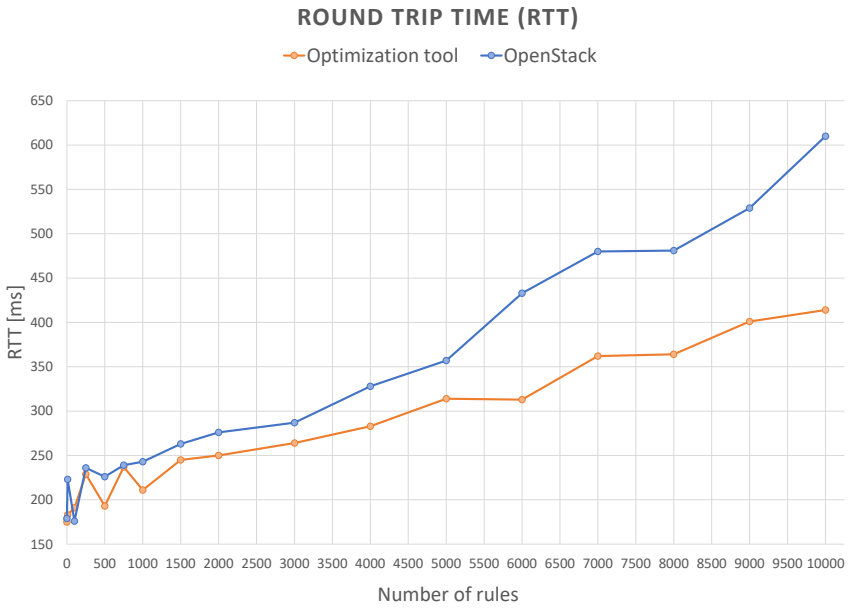

Figure 8.9. Latency test report

.

Figure 8.9 shows that performing a distribution over the filtering points according to the traffic pattern brings several benefits to the performances, reducing the latency due to the exploration of all the filtering rules in the Iptables chains.

Bandwidth test

The following Figure 8.10 shows that the bandwidth performance degrades as a negative exponential with the increase of the rule numbers. Using both OpenStack and the filtering tool do not change the performance trend, but it changes the maximum values that can be reached during the tests.

In particular it is visible that, using the distribution tool, the number of filtering rules that are injected into the single device became a quarter of the ones inject using OpenStack. The performance graph reflects precisely this behaviour: at `10000` rules the maximum bandwidth value obtainable using the distribution tool is the same as using OpenStack with `4000` rules.

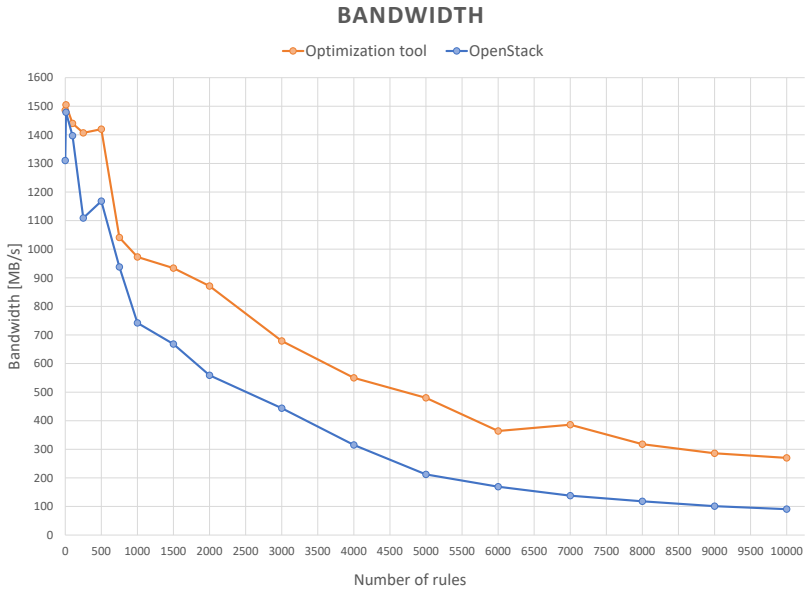

Figure 8.10. Bandwidth test report

In this test case is clearly visible how Iptables affects the performances, while OpenVSwitch do not contribute to them. Using the distribution tool, the rules will be injected also in the virtual switches composing the cloud infrastructure, but the high performances of these devices makes its contribute to the performance tests negligible.

8.4 Test conclusions

The testing phase showed how performing an optimized distribution could improve the performances of a cloud infrastructure. In particular it emerged that the first optimization can be done directly on the security policies, removing all the possible conflicts and reducing, consequently, the total number of rules to inject. If the number of rules in the security policy became higher a smart distribution process could reduce the performance degradation. Finally exploiting other technologies has demonstrated that it could lead to important results.

Chapter 9

Conclusions

This thesis work was part of a bigger project, realized with another student, with ambitious objectives in SDN and NFV research fields. It was necessary to focus on communication and collaboration because the two parts were complementary but independent, so integrate them together could become complicated and prone to errors.

The first phase of this work was the documentation of the background notions needed to understand the problem and start designing a solution. It was important to focus on what is the current "state-of-the-art" to design an innovative solution that can exploit modern technologies and overcome to business requirements of the market.

The research was not limited to the theoretic approach because the final goal was to build a tool, ready to be integrated in a framework. The main objective was to find a way to enhance the performance of security controllers over cloud infrastructures, simplify the management tasks done by the administrators, and providing an automatic configuration system for the security devices.

To achieve this objective, it was necessary to study how a cloud architecture works in detail. OpenStack was chosen as a study case because it better represent both the NFV and SDN concepts, in a simple and intuitive environment. Studying this Virtual Infrastructure Manager (VIM) was possible to identifying several filtering points across the infrastructure available for the security policy enforcement. Performing several tests on them was possible to identify their performance trend and to understand their behavior under heavy load situations.

In particular in OpenStack the security mechanism is handled by Security Groups that reflects security policies on the Iptables of the infrastructure physical nodes. Iptables is extremely easy to use, but its performances, when the number of rules starts to grow, are not enough to overcome the modern Cloud Computing requirements.

The study on the OpenStack infrastructure shows that it was possible to exploit the virtual switches used by the VIM for traffic forwarding between virtual machines, to convert them in filtering devices. Due to the design of OpenVSwitch devices, it is possible to obtain high performances in terms of network and resource consumption. This result was unexpected and it brought the research objective to find a way to exploit this technology for the main project goal.

In my thesis work, the objective was to implement the distribution module that, given the rules and the relative filtering points to inject them, loads the rules on the respective "virtual device".

Before implementing the distribution process was necessary to create a process to convert the security policies received as input into commands that can be executed on the devices. It was therefore necessary to perform a conversion mechanism that takes as input the single filtering rule and produces the set of commands (Iptables) or messages (OpenFlow) that needs to be launched on the relative filtering point.

The tool was validated using automated test, verifying both the efficiency of the optimization process and the performance gain relative to the rule distribution.

The test results proved that an optimization process of the cloud infrastructure could lead to a significant gain in performance. In particular, due to the OpenVSwitch design, it is possible to gain high performance, with no decreases despite the rise in number of filtering rules. The results confirmed the success of the research work and offers a set of key points on which focus in future works.

9.1 Future works

The results of this thesis work highlighted the importance of the optimization of a cloud infrastructure, moreover it brought different sections on which it is possible to concentrate future works and projects.

The tool was realized according to the *microservices* paradigm, building a series of independent modules. Each of these modules can be extended and integrated to other projects.

9.1.1 High level security

In this work, the objective was to find a way to exploit several filtering points to improve the performances of security controllers. The filtering functions are limited to the transport layer security. This choice was made because the main objective of the project was to find an efficient optimization and distribution model for a cloud infrastructure.

Future project may enhance high level security providing support for application level and stateful connections. To achieve this it necessary to extend the rule conversion modules adding the support for high level rules and policies.

It is important to mention that OpenVSwitch performs an enhanced and optimized packet filtering based only on the packet headers. This allows it to have high performances, because it uses hash based data structures and caches to store the flows and retrieve them based on the packets headers. However, it is not possible to perform high level traffic filtering on the virtual switches and it is necessary to use other security softwares.

Stateful packet filtering can be implemented both in Iptables and in OpenVSwitch devices. Both of them are based on the *conntrack* Linux kernel module that stores information about the state of a connection in a memory structure that contains the source and destination IP addresses, port number pairs, protocol types, state, and timeout. This module do not perform traffic filtering but can be exploited to perform filtering operations on the incoming traffic.

The stateful filtering discards all the packets not related to any existing TCP connection (that has completed the `SYN/SYN-ACK/ACK` procedure). This process discard all the packets with anomalous TCP flags, avoid common attacks such as port scanning or system fingerprinting.

9.1.2 Security softwares

Computing security offers a large number of applications and softwares that can be used to perform security operation. In the realized tool there is a support for Iptables because it is the application used by OpenStack to provide security fuctions to the infrastructure.

The Iptables module is completely independent from other modules and can be excluded if an infrastructure do not uses the same application to bring security features.

Therefore, it is possible to implement an additional module that implements the management of another application that offers different features. Squid is a caching proxy for the Web supporting HTTP, HTTPS, FTP, and more. It reduces bandwidth and improves response times by caching and reusing frequently-requested web pages [26]. It can be also used as a filter for

HTTP, HTTPS and FTP requests because it performs a deep packet inspection not possible with Iptables.

To add the support of another software it necessary to implement a module that, given a filtering rule as input, produces a set of commands that produces the expected result on the application. Once the corresponding commands are generated, the command dispatcher module can be used specifying the destination to launch the command on the target.

The module offers the possibility to launch commands locally or remotely using SSH, providing maximum flexibility and compatibility with every infrastructure. Is worthless to mention that to perform remote command injection it is necessary to have a fully operational SSH connection with authentication based on SSH keys to improve the security of the connection.

9.1.3 Cloud architectures

In the modern Internet, cloud computing technologies became a primary resource to deploy cloud services and applications. To deploy cloud application was necessary to develop cloud toolkits that allows the administrator to manage easily the architecture. These applications are called Virtual Infrastructure Manager (VIM) and they provide all the necessary services to manage virtual machines, virtual networks, databases, etc.

In this thesis work OpenStack is chosen as a reference on which focus the tool deployment. Due to its usability and elasticity it was rated as the best test bed on which perform the required researches. Nowadays there are several VIM available on the market (OpenStack, Kubernetes, Nebula, etc.) so it may be useful to extend the tool compatibility with others cloud operating systems.

To achieve this it is necessary to understand how is the internal infrastructure of the service that is going to be supported. Knowing how the traffic flows, how the virtual machines communicates each other or how they are interconnected is a key point to implement the features needed for the optimization process.

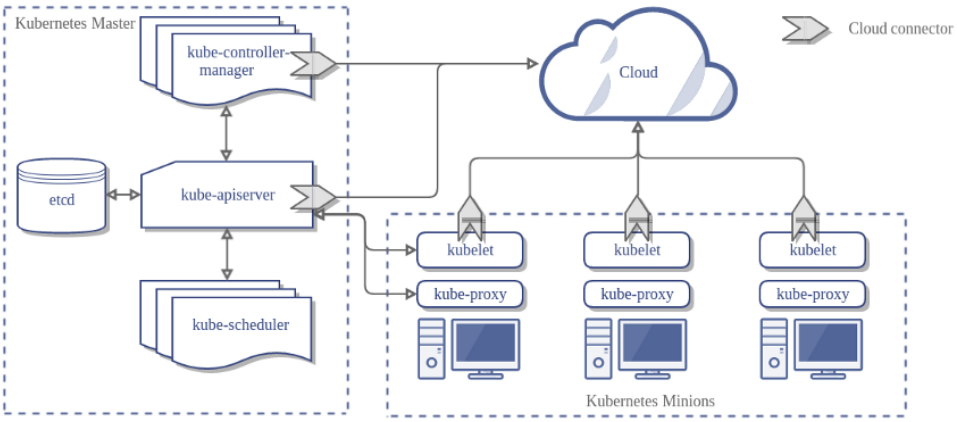

Figure 9.1. Kuberentes infrastructure reference scheme

According to Figure 9.1, in Kubernetes infrastructure there is a *master node* and several computing nodes (*minions*) [27].

Services are running in container inside the PODs, which can be virtual machines or physical hosts. Moreover K8s do not give any information on how the internal infrastructure should be designed, it only gives some constraints that need to be followed to ensure the correct behaviour of the toolkit.

Several projects deploy Kubernetes using an infrastructure similar to OpenStack, using virtual switches to interconnect the pods. These switches are often based on the OpenVSwitch project due to its extremely efficient behaviour also when the number of filtering rules became huge.

Is important to understand how the virtual machine are interconnected, if some virtual switch can be used as filtering point and eventually how to inject the security policy on the "devices".

Appendix A

Programmer manual

In this section the tool architecture and all its feature are explained in details, with a code function presentation for each part of the tool. The code language chosen for the orchestration tool is Java and also all the additional tools, are written in the same language, in this way they could be integrated. The input and the output data are XML files and validated by XSD schemas, they are handled by APIs provided by Java for the XML Processing (JAXB). The architecture presents REST APIs to perform input/output operations and to manage the program lifecycle. In this tool there are external libraries from different projects:

- Policy management, *it.polito.policytoollib*, to use directly the source code;
- Virtual switch managment, *floodlight.jar*;
- Server, *tomcat-dbcp.jar*, *tomcat-embed-core.jar*, *tomcat-embed-el.jar*, *tomcat-embed-jasper.jar*, *tomcat-embed-websocket.jar*.
- Resource dependency management, *hk2-api-2.6.1.jar*, *hk2-core-2.6.1.jar*, *hk2-locator-2.6.1.jar*, *hk2-utils-2.6.1.jar*;
- Jersey and JAXB, *jaxrs-ri-2.30.1.jar*;
- SAT Solver, *com.microsoft.z3.jar*

A.1 Physical topology

The physical infrastructure needs to be modelized inside the tool to have a clear view of how physical nodes, virtual switches and virtual machines are deployed in the environment. With this model all the physical parameters needed by the tool (e.g. IP addresses, usernames, virtual switch identifiers) are stored inside the respective objects. Every component of the deployment contains a rule dispatcher that is called every time a rule needs to be injected into the device.

A.1.1 Architecture

The model is represented by the package `it.polito.policyorchestration.impl.topology.physical` and contains an abstract class that represents the filtering points in the infrastructure, extended by several sub-classes.

- `class Deployment` contains all the physical hosts and represents the infrastructure which the tool is running on. One or more virtual switch are stored inside `PhysicalHost` object, and every virtual switch contains a list of `ServiceNode` objects. A `ServiceNode` is an object that represents the infrastructure element which contains one or more services (e.g. Virtual-Machines, POD). A deployment is characterized by a Virtual Infrastructure Manager (VIM) (e.g. OpenStack, Kubernetes, etc.). The `Deployment` object has the following methods:

  - `public LinkedList<PhysicalHost> getPhysicalHostList()`
    returns the list of `PhysicalHost` objects loaded in the deployment;
  - `public PhysicalHost searchPhysicalHost(String ip)`
    returns the physical host with the given IP address, or `null` if the host is not present in the deployment;
  - `public LinkedList<VirtualSwitch> getVirtualSwitches()`
    returns the list of `VirtualSwitch` objects loaded in the deployment;
  - `public VirtualSwitch searchVirtualSwitch(String label)`
    returns the virtual switch with the given *datapath-id* (label), or `null` if the virtual switch is not present in the deployment;
  - `public LinkedList<ServiceNode> getServiceNodes()`
    returns the list of `ServiceNode` objects loaded in the deployment;
  - `public ServiceNode searchServiceNode(String ip)`
    returns the service node with given IP address, or `null` if the service node is not present in the deployment;
  - `public ServiceNode removeServiceNode(String ip)`
    removes the service node with given IP address from the the deployment, it returns the removed service node, or `null` if the service node is not present in the deployment;
  - `public void setFloodlightRuleDispatcher(FlowDispatcher dispatcher)`
    sets, for each virtual switch, the Floodlight flow dispatcher module to inject OpenFlow flows;
  - `public void initializeIptables()`
    initialize, for each physical node and service node, the Iptables chain where rules will be injected after distribution process;
  - `public void cleanup()`
    removes, for each physical node and service node, the Iptables rules injected by the tool;
  - `public void resetFlags()`
    resets the `isExcluded` and `isMiddlebox` flags of the `FilteringPoints` that compose the deployment;
  - `public LinkedList<FilteringPoint> getFilteringPoints()`
    returns the list of `FilteringPoints` that compose the deployment.

- `abstract class FilteringPoint`: this class is an abstract class, it represents a device in which rules can be inserted. It contains all the variables that are in common between all the filtering points of the environment. The shared variables are:

  - `private String identifier`
    is the unique identifier of the filtering point, it can be an IP address for `ServiceNode` and `PhysicalHost` objects, or it can be the *Datapath-id* of a `VirtualSwitch`;
  - `private PerformanceTool performance`
    is the performance tool used to compute the performance rating of the filtering point, according to the number of rule that will be injected on it. The performance rating is used by the distribution tool to find the proper device to inject rules;
  - `private boolean isExcluded`
    indicates if the filtering points is excluded from the rule distribution process;
  - `private LinkedList<GenericRule> ruleList`
    contains the list of rules loaded into the device;
  - `private Action defalutAction`
    represents the default action of the filtering point, it is set based on the entity policy default action;

  `FilteringPoint class`, other than default getters and setters, has the following methods:

- `LinkedList<GenericRule> addRuleList(LinkedList<GenericRule> ruleList)`
  add all the of the list passed as parameters to the object' rule list. If the number of rule exceed the maximum number that the physical host can support, the maximum number of rules will be inserted. This methods returns `null` if all rules were inserted correctly, otherwise it return a list containing the exceeding rules;
- `public void addRule(GenericRule rule)`
  adds a single rule to the object rule list;
- `public void printInjectableRules()`
  prints all the rules that are loaded inside the object;
- `public abstract void injectRules()`
  starts the rule injection process, the rule injection process is different accordingly to the filtering point type.
- `public int compareTo(FilteringPoint fp)`
  overrides the default comparator for `FilteringPoint` objects, the result depends of the performance rating given by the performance tool;

This class is extended by three classes that represent the different types of filtering point that the tool supports:

- `class PhysicalHost`: this class represents a physical host in the infrastructure, it is identified by an IP address. It contains the list of virtual switches, where service nodes will attach to communicate. Each `PhysicalHost` has a `CommandDispatcher` object that contains all the required libraries to dispatch system commands locally or via SSH connections (e.g. Iptables rule injection commands). `PhysicalHost` class has the following methods:
  - `public void startup()`
    initializes the physical host creating a new Iptables chain, where rules will be inserted. After chain creation it inject a rule in existing chains to redirect all traffic to the newly created one;
  - `public void shutdown()`
    removes all injected rules and created chains, restoring previous state;
  - `public void addVirtualSwitch(VirtualSwitch sw)`
    adds a new `VirtualSwitch` object to the virtual switch list;
  - `public String getSecurityUserName()`
    returns the username needed to log into the machine from remote hosts. This user must be enabled to run Iptables commands inside the machine.
- `class VirtualSwitch`: this class represents a virtual switch where service nodes will be attached. Every virtual switch has a rule dispatcher that will inject filtering rules. In case of OpenFlow switches it uses the Floodlight library to create custom flows and inject them into the switch. `VirtualSwitch` class has the following methods:
  - `public void addServiceNode(ServiceNode n)`
    adds a `ServiceNode` object to the virtual switch node list;
  - `public ServiceNode removeServiceNode(ServiceNode n)`
    removes a service node from the list. It returns the removed `ServiceNode` object or `null` if the object is not present in the list.
  - `public VSWTypeElement getType()`
    returns the type of the virtual switch (e.g. *OVS*, *LINUXBRIDGE*, etc.);
- `class ServiceNode`: this class represents the element in the infrastructure in which services are running (e.g. a virtual machine, a Kubernets pod, etc.). It is identified by an IP address and contains the list of services that are running on it. Like physical hosts it contains a `CommandDispatcher` object for rule loading. `ServiceNode` class has the following methods:
  - `public void startup()`
    initialize the physical host creating a new Iptables chain, where rules will be inserted. After chain creation it inject a rule in existing chains to redirect all traffic to the newly created one;

  - `public void shutdown()`
    removes all injected rules and created chains, restoring previous state;
  - `public void setMiddleBox(boolean isMiddleBox)`
    marks the service node as a middlebox. A middlebox is a service node capable of traffic routing in which is possible to apply traffic filtering policies;
  - `public boolean isMiddleBox()`
    returns `true` if the service node is also a middlebox, `false` otherwise;
  - `public void addService(Service s)`
    adds a `Service` object to the services list;
  - `public void deleteService(String port)`
    removes a service from the list. It returns the removed `Service` object or `null` if the object is not present in the list.
  - `public NodeTypeElement getType()`
    returns the type of the service node (e.g. *VM*, *POD*, etc.);

- `class PhysicalTopologyFactory`: is the class that manage all physical topology object creation and initialization. It can not be instantiated bacause it has private constructor. `PhysicalTopologyFactory` class has the following methods:

  - `public static Deployment createDeployment(DeploymentElement dep)`
    creates a new `Deployment` object from a `DeploymentElement`. The `DeploymentElement` object is the result of XML parsing process done by `JAXB Framework`. This method validates the element passed as parameter, using the `PhysicalTopologyValidator` class and if the element is correct starts to create the `Deployment` adding physical hosts, virtual switches and service nodes. It also instantiate the `PerformanceToolFactory` object;
  - `public static PhysicalHost createPhysicalHost(Deployment dep, HostElement host, PerformanceToolFactory pf)`
    creates, after proper element validation, a new `PhysicalHost` object from a `HostElement`;
  - `public static VirtualSwitch createVSwitch(Deployment dep, VSwitchElement vswitch, PerformanceToolFactory pf)`
    creates, after proper element validation, a new `VirtualSwitch` object from a `VSwitchElement`;
  - `public static ServiceNode createNode(Deployment dep, NodeElement serviceNode, PerformanceToolFactory pf)`
    creates, after proper element validation, a new `ServiceNode` object from a `NodeElement`;

A.1.2 Modify the tool

It is possible to extend the `FilteringPoint` class adding new types of filtering point. Currently the tool is supporting physical hosts, virtual switches and service nodes as filtering point. It is necessary to implement the `void injectRules()` method, defining how rules will be inserted into the defined filtering point. It will be necessary to add the element to the XML and extend the input reading module.

A.2 Distribution tool

The model tool is represented by the package `it.polito.policyorchestration.impl.optimization.distribution` and it splits one equivalent firewall into many different distributed firewall in a physical and virtual deployment. It distribute the rules only on the nodes where the traffic could flows and it minimize the number of rules injected using the performance tool. Distribution tool produce the output follows the workflow:

- reads the rules to be injected;
- reads the available filtering points where the rules could be injected;
- for each rule compute all the possible path from each destination to each sending machines;
- for each node on the path set that is a possible candidate to host the rule;
- create all the equation to be passed to a SAT solver;
- distribute the rule on each chosen filtering point.

A.2.1 Tool architecture

The tool architecture is made up of a class that handles the distribution of the rules. It exploits the features of another class used to represent the deployment architecture through an interface and the performance tool to optimized the distribution. The main class structure is the following:

- `RuleDistributorTool` is the class that handles the rule distribution, it knows only the set of rules and the set of filtering points associated to each rule. The class present the following methods:
  - `public RuleDistributorTool(InfrastructureManager tool)`
    is the constructor that has an InfrastructureManager as parameter, in which there are the rules and all the informations used to optimized the distribution;
  - `public void injectRules()`
    is the method that, for each filtering point available, calls the methods that inject the rules associated into it self;
  - `private void findAllTheRulesForEachFilteringPoint()`
    is a method that for each rule find a path, that means fining all the filtering points, between the receiving point and the transmission point. In other words a rule could be matched by a traffic pattern, this method find all the destination points for this type of traffic and for each one find where the rule must be placed or could be place in a such way that the number of filtering point chosen is the minimum;
  - `private void createBooleanVerticalEquationsForTheModel()`
    is a method that builds a boolean equation for each filtering point. The purpose of this equation is set a variable $\mathcal{P}_{\mathrm{n,i}}$, that could be *true* or *false*, and represents if the rule n must be injected on the filtering point i;
  - `private void createBooleanHorizontalEquationsForTheModel()`
    is a method that builds a boolean equation for each rule. The purpose is imposed that if a rule, which as the same action of the default action, could be injected must be injected but at least one time;
  - `private void optimizeModel()`
    is a method that create a real cost equation by sum all the performance of each filtering point. At the end this function is minimize and the model is compute;
  - `private void distributeRulesAsModel()`
    is a method that read the result of the computation of the optimizer and set for each filtering points their rules.

- `DeploymentStructure`: it is the interface for the deployment structure that is independent from the environment, it has two methods:
  - `LinkedList<DeploymentNode> getAllFilteringPoints()`
    returns all the filtering point of the deployment structure
  - `public LinkedList<TreeNode> getConditionClausePoints(ConditionClause c)`
    is a method that is useful to retrieve one or more filtering points and end-points into the tree structure.
- `TreeStructure`: it is the class that handles the structure of the deployment and implements the `DeploymentStructure` interface, it is in charge of modelling the physical and the virtual points and create a tree where they are placed. This class has the following methods:
  - `public TreeStructure(InfrastructureManager tool, LinkedList<IpSelector> internalSubnetList)`
    is the constructor of the class, the parameters are the `InfrastructureManager tool` used to retrieve the physical and virtual environment and the `LinkedList<IpSelector> internalSubnetList` that is a list of all the subnet present inside the deployment. This last parameter is useful to understand if the traffic pattern is from or to external points or internal ones;
  - `private void populateTree()` is the method that build the tree with the information of the `InfrastructureManager tool`;
- `DeploymentNode`: it is the interface for the deployment node that is independent from the deployment structure used, it has two important methods:
  - `LinkedList<String> getORPointPath(ConditionClause arrival, ConditionClause departure, IpSelector subnet)`
    is the method that finds the minimal set of possible filtering points from the end-point `arrival` to any `departure` points. The last parameter `subnet` identify the entity that own the rule for which the method try to find a path for the corresponding traffic pattern. This method is based on ten models (see Appendix **??**) that the single node could be, it chooses what kind of model it is;
  - `public LinkedList<String> getANDPointsPath(ConditionClause arrival, ConditionClause departure, IpSelector subnet)`
    is the method that finds the minimal set of necessary filtering points from the end-point `arrival` to any `departure` points. The last parameter `subnet` identify the entity that own the rule for which the method try to find a path for the corresponding traffic pattern. This method is based on ten models (see Appendix **??**) that the single node could be, it chooses what kind of model it is;
- `TreeNode`: it is the class that represent a single node inside the tree structure and implements the `DeploymentNode` interface. All the TreeNode that are internal nodes are filtering points, on the contrary the leaf nodes are the services with an IPaddress and a Port associated. The class has the following instance variables:
  - `private String label`
    identifies the node;
  - `private TreeNode parent`
    is the variable pointer to the parent node, this is always not `null` except for the root element;
  - `private List<TreeNode> children`
    is the list of the pointers to all the children of the node, this list is always `notEmpty` except for the leaf elements;
  - `private FilteringPoint filteringPoint`
    is the pointer to the physical or virtual filtering point of the `InfrastructureManager`;

- `private LinkedList<ConditionClause> childConditionClauseList`
  is the list of all the end-points under this node, that means all the end-points that are reachable from this node passing through its children;
- `private LinkedList<ConditionClause> fatherConditionClauseList`
  is the list of all the end-points above this node, that means all the end-points that are reachable from this node passing thought its parent.

- `public TreeNode(FilteringPoint filteringPoint, boolean isGateway, ConditionClause externalConditionCaluse)`
  is the first constructor of the class, `filteringPoint` is the element of the deployment, `isGateway` identify if this node reach directly the exterior of the deployment, `externalConditionCaluse` represent all the destination that are external to the deployment;
- `public TreeNode(String label, boolean isService, boolean isRoot, ConditionClause externalConditionCaluse)`
  is the second constructor of the class, it is useful because both root element and leaf elements are not related to any `filteringPoint` or could not be gateway;
- `public void addChildConditionClause(ConditionClause c)`
  `public void addChildConditionClauseList(LinkedList<ConditionClause> cl)`
  `public LinkedList<ConditionClause> getChildConditionClauseList()`
  `private boolean hasChildrenConditionClause(ConditionClause c)`
  they are methods that manage the instance variable `childConditionClauseList`;
- `public void addFatherConditionClauseList(LinkedList<ConditionClause> cl)`
  `public LinkedList<ConditionClause> getFatherConditionClauseList()`
  `private boolean hasFatherConditionClause(ConditionClause c)`
  they are methods that manage the instance variable `fatherConditionClauseList`;

A.2.2 Modify the tool

It is possible to change the model tree of the deployment with other model, as chain or graph. Currently it supports only the tree model because it is the mostly used physical and logical topology that avoids possible loops, besides that the work related to this thesis is based on an OpenStack deployment. To change the model representation must add a new class that substitutes the `TreeStructure class` and implements the `deploymentStructure` interface. Then a new class is needed to substitute the `TreeNode class` that implements the `deploymentNode` interface. All the interface's methods must be implemented following the previous definition.

A.3 RESTful web service

The RESTful web service allows the users to store and retrieve informations about the infrastructure on which our tool is operating. It permits to manage physical/logical topology and filtering policies given by the security administrator.

The REST resource classes are implemented in the `it.polito.policyorchestration.rest.resources` package and all the classes in the package will be loaded into the *Jersey* servlet at startup.

To manage HTTP requests, a `PolicyOrchestrationService` object is injected into resources every time a REST method is called. It is responsible to maintain synchronization between requests, avoiding race conditions.

A.3.1 Architecture

The `PolicyOrchestrationService` contains the `InfrastructureManager` object and the `LifecycleManager` as attributes. The former contains all the object that represent the infrastructure (e.g. `Deployment`, `Landscape`, `Entities`, etc.), the latter offers the methods to manage the lifecycle of the program.

This class offers the following methods:

- `public PolicyOrchestrationService(InfrastructureManager infrastructureManager)`
  is the constrcutor method that has `InfrastructureManager` as parameter in input;
- `public synchronized void start()`
  starts the rule distribution process. It starts with an initialization of all the infrastructure elements (e.g. physical hosts, virtual switches, etc.) inserting some traffic redirection to be able to perform filtering operations. The it starts the distribution process, optimizing rule positions and finally inserting them;
- `public synchronized void shutdown()`
  closes the program with the cleaning of all the rules injected;
- `public synchronized void setDeployment(Deployment d)`
  `public synchronized void setLandscape(Landscape l)`
  `public synchronized void setSelectorTypes(SelectorTypes s)`
  `public synchronized void setEntities(Entities e)`
  these methods are the "setters" for the `InfrastructureManager Object`;
- `public synchronized void getDeployment(Deployment d)`
  `public synchronized void getLandscape(Landscape l)`
  `public synchronized void getSelectorTypes(SelectorTypes s)`
  `public synchronized void getEntities(Entities e)`
  these methods are the "getters" for the `InfrastructureManager Object`;
- `public synchronized DeploymentElement getDeploymentElement(boolean onlyNetworkNodes, UriInfo root)`
  creates the `DeploymentElement` from the loaded `Deployment`, there are also the same method for `PhysicalHostElement`, `VirtualSwitchElement` and `ServiceNodeElement`;
- `public synchronized Deployment loadDeployment(DeploymentElement root)`
  initializes the `Landscape` object;
- `public synchronized LandscapeElement getLandscapeElement(String type, UriInfo root)`
  creates the `LandscapeElement` from the loaded `Landscape`, there are also the same method for `LandscapeHostElement` and `LandscapeServiceElement`;
- `public synchronized Landscape loadeLandscape(LandscapeElement net)`
  initializes the `Landscape` object;

- `public synchronized EntitiesElement getEntitiesElement(boolean onlyISP, UriInfo root)`
  creates the `EntitiesElement` from the loaded `Entities`, there are also the same method for `EntityElement` and `RuleElement`;
- `public synchronized Entities loadEntities(EntitiesElement ent)`
  initializes the `Entities` object;
- `public synchronized FilteringPoints getFilteringPoints(String type, UriInfo root)`
  creates the `FilteringPointsElement` from the loaded FilteringPoints, there are also the same method for `ExclusionElement` and `MiddleBoxElement`;
- `public synchronized void loadFilteringAndExclusions(FilteringPoints fp)`
  updates the environment according to file passed as parameter, it set filtering point (VM that acts as router and packet filter) and it sets the excluded physical topology elements;
- `public synchronized DistributionOutcome getDistributionOutput()`
  returns the output of the distribution process.

A.3.2 Modify the tool

To implement new REST methods, it is necessary to add an API on the preferred resource path. All the operation on the `InfrastructureManager` data should be executed in `PolicyOrchestrationService` class in a `synchronized` context to preserve the correct concurrency. For example, if we want to implement methods to add/remove a virtual switch from the `Deployment` element, we need to add a method in the `DeploymentResources` class, that handles the `PUT`, `DELETE` requests. Then is necessary to implement in the `PolicyOrchestrationService` class the method to add/remove a virtual switch from the `Deployment` element loaded in `InfrastructureManager` object.

Appendix B

REST APIs

All the resources are under the root folder `api/`

Resource	Method	Path	Description
`Deployment`	`GET`	/deployment	Read physical topology informations
	`GET`	/deployment/{IP}	Read a physical host informations
	`GET`	/deployment/{IP}/{label}	Read a virtual switch informations
	`GET`	/deployment/{IP}/{label}/{nodeIP}	Read a service node informations
	`POST`	/deployment	Load physical topology structure
	`PUT`	/deployment/{IP}/{label}/{nodeIP}	Update or create a service node
	`DELETE`	/deployment/{IP}/{label}/{nodeIP}	Delete an existing service node
`Landscape`	`GET`	/landscape	Read logical topology informations
	`GET`	/landscape/{nodeIP}	Read a service node informations
	`GET`	/landscape/{nodeIP}/{port}	Read a service informations
	`POST`	/landscape	Load logical topology structure
	`PUT`	/landscape/{nodeIP}	Update or create a service node
	`PUT`	/landscape/{nodeIP}/{port}	Update or create a service
	`DELETE`	/landscape/{nodeIP}	Delete an existing service node
	`DELETE`	/landscape/{IP}/{port}	Delete an existing service
`Entities`	`GET`	/entities	Read entities informations
	`GET`	/entities/{label}	Read an entity informations
	`GET`	/entities/{label}/{rulelabel}	Read a filtering rule informations
	`POST`	/entities	Load entities structure
	`PUT`	/entities/{label}	Update or create an entity
	`PUT`	/entities/{label}/{rulelabel}	Update or create a filtering rule
	`DELETE`	/entities/{label}	Delete an existing entity
	`DELETE`	/entities/{label}/{rulelabel}	Delete an existing filtering rule
`FilteringPoints`	`GET`	/filteringpoints	Read middleboxes and exclusions informations
	`POST`	/filteringpoints	Load middleboxes and exclusions structure
	`PUT`	/filteringpoints/middleboxes/{nodeIP}	Add a service node to the middleboxes list
	`DELETE`	/filteringpoints/exclusions/{id}	Add an element to the exclusions list
	`DELETE`	/filteringpoints/middleboxes/{nodeIP}	Remove a service node to the middleboxes list
`SelectorTypes`	`GET`	/selectors	Read selector types informations
	`POST`	/selectors	Load selector types structure
`Tool`	`GET`	/tool/output?type=	Read distribution outcome informations
	`OPTIONS`	/tool/start	Starts rule distribution procedure
	`OPTIONS`	/tool/stop	Starts tool shutdown procedure

Table B.1. Available REST APIs

Bibliography

[1] Bo Yi, Xingwei Wang, Keqin Li, Sajal k. Das, Min Huang, "A comprehensive survey of Network Function Virtualization", Computer Networks, Vol. 133 May 2018, pp. 212-262, DOI 10.1016/j.comnet.2018.01.021

[2] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, Raouf Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges", IEEE Communications Surveys & Tutorials, Vol. 18, No. 1, Firstquarter 2016, pp. 236-262, DOI 10.1109/COMST.2015.2477041

[3] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, Seungjoon Lee, "Network function virtualization: Challenges and opportunities for innovations", IEEE Communications Magazine, Vol. 53, No. 2, February 2015, pp. 90-97, DOI 10.1109/MCOM.2015.7045396

[4] ETSI, "Network Functions Virtualisation (NFV): Architectural Framework", [Online], https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf

[5] Open Networking Foundation (ONF), "Software-Defined Networking (SDN) Definition", [Online], https://www.opennetworking.org/sdn-definition/

[6] National Institute of Standards and Technology (NIST), "The NIST Definition of Cloud Computing", [Online], https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[7] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, Haiyong Xie, "A Survey on Software-Defined Networking", IEEE Communications Surveys & Tutorials, Vol. 17, No. 1, Firstquarter 2015, pp. 27-51, DOI 10.1109/COMST.2014.2330903

[8] Internet Engineering Task Force - IETF, "Interface to Network Security Functions (i2nsf)", [Online], https://datatracker.ietf.org/wg/i2nsf/about/

[9] Ehab S. Al-Shaer, Hazem H. Hamed, "Firewall Policy Advisor for anomaly discovery and rule editing", IFIP/IEEE Eighth International Symposium on Integrated Network Management, Colorado Springs (USA), 2003, pp. 17-30, DOI 10.1109/INM.2003.1194157

[10] Hazem H. Hamed, Ehab S. Al-Shaer, "Taxonomy of conflicts in network security policies", IEEE Communications Magazine, Vol. 44, No. 3, March 2006, pp. 134-141, DOI 10.1109/MCOM.2006.1607877

[11] Ehab S. Al-Shaer, Hazem H. Hamed, "Discovery of policy anomalies in distributed firewalls", IEEE INFOCOM 2004, Vol. 4, Hong Kong, 2004, pp. 2605-2616, DOI 10.1109/INFCOM.2004.1354680

[12] Hazem H. Hamed, Ehab S. Al-Shaer, "Dynamic Rule-Ordering Optimization for High-Speed Firewall Filtering", Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, Taipei (Taiwan), March 2006, pp. 332-342, DOI https://doi.org/10.1145/1128817.1128867

[13] Hazem H. Hamed, Adel El-Atawy, Ehab S. Al-Shaer, "On Dynamic Optimization of Packet Matching in High-Speed Firewalls", IEEE Journal on Selected Areas in Communications, Vol. 24, No. 10, October 2006, pp. 1817-1830, DOI 10.1109/JSAC.2006.877140

[14] David D. Clark, David R. Wilson, "A Comparison of Commercial and Military Computer Security Policies", 1987 IEEE Symposium on Security and Privacy, Oakland (USA), 1987, pp. 184-184, DOI 10.1109/SP.1987.10001

[15] Fulvio Valenza, Manuel Cheminod, "An Optimized Firewall Anomaly Resolution", Journal of Internet Services and Information Security (JISIS), Vol. 10, No. 1, February 2020, pp. 22-37, DOI 10.22667/JISIS.2020.02.29.022

[16] Cataldo Basile, Alberto Cappadonia, Antonio Lioy, "Network-Level Access Control Policy Analysis and Transformation", IEEE/ACM Transactions on Networking, Vol. 20, No. 4, August 2012, pp. 985-998, DOI 10.1109/TNET.2011.2178431
[17] OpenStack project, https://www.openstack.org/software/
[18] Linux top man page, http://man7.org/linux/man-pages/man1/top.1.html
[19] Nping project, https://nmap.org/nping/
[20] iPerf3 project, https://iperf.fr/
[21] Iptables project, https://linux.die.net/man/8/iptables
[22] Open vSwitch project, https://www.openvswitch.org/
[23] JSch project, http://www.jcraft.com/jsch/
[24] ovs-ofctl command man-page, https://www.systutorials.com/docs/linux/man/8-ovs-ofctl/
[25] Floodlight project, https://floodlight.atlassian.net/wiki/
[26] Squid project, http://www.squid-cache.org/
[27] Kubernetes project, https://kubernetes.io/it/
[28] Global public cloud computing market 2008-2020, https://www.statista.com/statistics/510350/worldwide-public-cloud-computing/