Development of a Cyber Range with description language for network topology definition

Master of Science Thesis
University of Turku
Department of Computing
Cyber security
2023
Dalla Costa Andrea

Abstract

Cyber Ranges are an essential tool for cybersecurity trainings and experiments because they enable to setup virtual, isolated and reproducible environments that can be safely used to execute different types of tests and scenarios. The preparation of scenarios is the most time-consuming phase, which includes the configuration of the virtual machines and the definition of the network topology, so it is important for a Cyber Range to include tools that simplify this operation. This work focuses on how to implement and setup a Cyber Range that includes the necessary features and tools to simplify the setup phase, in particular for large topologies. The literature review provides an analysis of the selected open-source and research solutions currently available for Cyber Ranges and their configuration for use in different scenarios. This work presents the development of a Cyber Range based on the open-source framework OpenStack and the entire design process of a new Description Language, starting from the analysis of the requirements for the defined use-cases, defining and designing the required features, the implementation of all the required components, and the testing of the correctness and effectiveness of the whole system. A comparison of the implemented solution against the selected solutions in the literature study is provided, summarising the unique features offered by this approach. The validation of the Description Language implementation with the defined use cases demonstrated that it can reduce the complexity and length of the required template, which can help to make the setup of scenarios faster.

Preface

I would like to thank Rautila Mika for the initial supervision and topic definition, Kuusijärvi Jarkko for the supervision and support during my work. I would also like to thank Hakkala Antti for the support and feedbacks on the thesis.

This thesis was written at VTT Technical Research Centre of Finland at the Safe and connected society research area as part of VTT's work in the EU project CyberMAR. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 833389. The content of this document reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Lastly, I would like to thank all my friends who supported me during the process of writing this thesis.

Contents

1 Introduction

The growing number of cyber-attacks is increasing the demand for security equipment and professionals, which makes testing and hand-on trainings on realistic networks a key aspect. Recreating networks using physical devices is very complex, expensive and difficult to setup, thus virtualized environments are often preferred, making Cyber Ranges are a popular solution because they provide virtualized and isolated environments that are easier and cheaper to setup and manage than replicating networks using real equipment. Moreover, Cyber Ranges allow to configure, deploy and reset the entire network structure automatically, simplifying the operations required to reproduce multiple trainings.

Despite being easier to setup, the planning and configuration of the scenario of training or simulation exercises is still the most time-consuming phase and usually requires the operators to define the network topology (or topology later in the text) and the various configurations of all the machines present in the virtual network. In our work we focus on the creation of a Cyber Range that makes use of a templating system that aims to simplify the topology definition process, by using a Description Language that we designed.

Chapter 2 provides a more complete description of Cyber Ranges and introduces OpenStack, the virtualization environment that we used as a base for our Cyber Range. Then existing Cyber Ranges and their templating systems are presented in Chapter 3. In Chapter 4 we provide the specific requirements that our solution

must be able to support and we present the grammar structure of our language. Chapter 5 shows the implementation of the compiler for our language and also of the other components that we developed. Our implementation is then tested in Chapter 6, to verify that the requirements identified before are fulfilled and it is also compared to other existing solutions. Lastly, Chapter 7 presents the conclusions and future work.

2 Background

In recent years, the number of cyber-attacks is constantly growing also thanks to the increasing number of devices connected to the internet. Moreover, the magnitude of their impact is more and more significant with more sensible data being store digitally. For example, the recent attack to Vastaamo, a Finnish psychotherapy service provider, resulted in the distribution of the highly sensible data related to the patients online [1].

In order to properly defend the IT infrastructure, assets and data from criminals, multiple technologies and solutions have been developed, with a notably increasing trend also in the research effort, with a growing number of published papers [2]. However, implementing and deploying security equipment in the network infrastructure is not sufficient, it is also necessary to both properly train security professionals and raise awareness among all employees on how to respond to possible cyber-attacks.

Therefore, simulations and hand-on exercises are key components to test and validate security equipment and to provide a realistic but safe environment for trainings. The setup of the system is not trivial, since it is usually not possible to use the real network in order to avoid disruptions to the normal operations. A possible solution is to replicate the network, setting up a dedicated environment that acts as clone of the real on, using the same devices and configurations. This solution is quite problematic for multiple reasons. Firstly, it is very expensive to recreate a large network, also in terms of time required to configure it. Secondly, resetting the

network to a clean state to start a new exercise or simulation can be difficult.

Another option is to virtualize or simulate part or the entirety of the testing environment, which alleviates many of the problems regarding the complexity of the setup and management of the network. These virtual environments are called Cyber Ranges.

2.1 Cyber Ranges

Cyber Ranges provide an isolated, reproducible and controlled environment that can be programmatically created, which can be used to emulate or simulate various kind of devices and configurations. Cyber Ranges make extensive use of virtualization technologies and use different stacks and different technologies to manage the virtualization of the devices and networks, depending on the goal. Physical devices can also be integrated in the virtual scenario. The technology used influences the accuracy of the simulated environment, its adaptability and performance.

Different virtualization technologies provide advantages in different aspects. **Hypervisor** based virtualization offers the best isolation and fidelity but requires the hypervisor module to be present in the kernel, which introduces some overhead. **Host** virtualization does not require a kernel module and provides good isolation but has a much higher overhead. Lastly, **container** technology, such as Docker, LXD or Podman, has very low overhead but also a much lower level of isolation. A good comparison of the three virtualization technologies is provided in [3], with a focus on the performance and overhead of the different solutions.

Moreover, some Cyber Ranges have the possibility to integrate also physical devices within the virtual environment. This is very useful in specific uses-cases, when the fidelity of a specific device is a very important aspect. However, using real devices reduces the flexibility of the system, resulting in a Cyber Range that is more difficult to manage, change or scale.

In recent years the interest in the topic has been increasing and many researches focused on implementing and improving the state of the art of Cyber Ranges, with focus on different aspects. There exist many literature review that summarize the advancement produced, and in particular [4], [5] provide a good view on the current status of Cyber Ranges development and also show the increasing number of papers published on the topic.

Cyber ranges can be used for different use-cases, depending on their implementation and the features they provide, but security testing and trainings are the most common ones. For **security testing** it is important to have an isolated environment, especially when potentially dangerous software is tested, such as malware. Moreover, being able to reset the environment to a clean state and re-perform the tests with different configurations or parameters is an essential feature. Moreover, the possibility to execute multiple test in parallel can produce faster and more significant results.

In the case of **trainings**, Cyber Ranges are a great tool to develop skills in multiple areas of cybersecurity [6]. For the exercises, often a virtual representation of a real network is deployed in the Cyber Range and the participants can be divided between red, attackers, and blue, defenders, teams. This allow to gain significant experience without the need to use and disturb the real network. Moreover, the ability to recreate large and different virtual scenario is clearly a key feature.

In fact, Cyber Ranges usually provide a *templating system*, which is used to create a configuration of the virtual environment, including the simulated network topology, the software used and other details, that can then be used to deploy and execute the scenario multiple times. The templating component is implemented differently depending on the use-cases and focus of the Cyber Range, and varies from graphical interfaces applications to configuration text files.

A good templating system allows the user to easily define all the required aspects

of the scenario, making the planning phase of exercises simpler. In fact, when considering security trainings, the preparation phase of the exercise is usually what requires most time and effort [7], [8]. Therefore, there is still the need to develop tools that can reduce the complexity and time spent on preparing trainings [9].

2.2 OpenStack

OpenStack is an open source project that can be used for creating public and private cloud computing platforms [10]. It enables the automatic deployment and configuration of virtual machines, virtual networks and other type of resources, with the possibility to distribute them between multiple physical servers, without manual intervention. This makes it quite easy to scale the infrastructure capabilities by adding additional dedicated or generic servers to extend the virtual environments capabilities. The scalability is achieved thanks to the OpenStack structure, which is composed of multiple components that can be installed and scaled independently.

Due to the feature that it provides, OpenStack is also a good base infrastructure to use for the development of a Cyber Range, as it automates many of the tasks required by a Cyber Range, as shown in Section 4.3. Therefore, many Cyber Ranges are developed using OpenStack as the underlying platform, as shown in Chapter 3.

The main components of OpenStack are described in the following Section and the networking component is analysed more in detail in Section 2.2.2. Then Section 2.2.3 shows some of the limitations of OpenStack, especially when it is used as a base infrastructure for developing a Cyber Range.

2.2.1 OpenStack components

OpenStack is composed of multiple components that interacts with each other over HTTP APIs and messages exchanged over the `RabbitMQ` [12] broker. This result

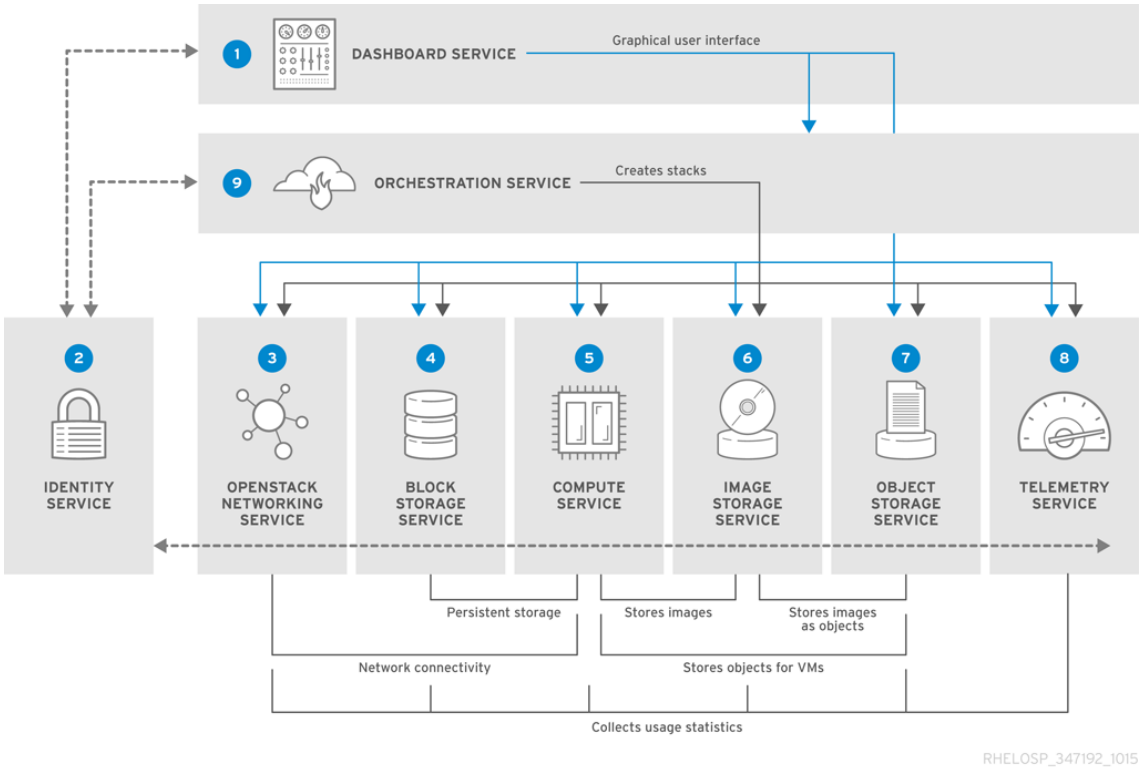

Figure 2.1: OpenStack components [11]

in a modular system, that can be adapted and customized to the requirements of different use-cases, by configuring or installing different and additional components.

The main components, that are usually present in every OpenStack installation, are the following [13] and are shown also in Figure 2.1 with their interactions:

- **Keystone** is the identity service of OpenStack and act as authentication point for both the users and the other components.
- **Nova** is the component that manages the virtual machines. The physical servers that have Nova installed are usually called *compute nodes*, since Nova create, start and stops the virtual servers utilizing different virtualization technologies, such as `QEMU`, `KVM`, `Hyper-V` and others. When multiple compute nodes are available, Nova also decides which physical node is the best suited for hosting a new virtual server, depending on the resources available on each node.
- **Neutron** manages the networking aspects of the virtual environments. It controls the virtual networks that are used by the virtual servers and provides other networking services, e.g. DHCP, Firewalls and other. Usually neutron is installed on every compute node but there might be the need to deploy be one or more dedicated networking nodes, depending on the requirements of the cloud platform. The next section will provide more details on how virtual networks are created and handled.
- **Glance** is used for storing images used by the virtual machines, together with additional metadata.
- **Cinder** is the *Block Storage* service that is used mainly for the volumes of virtual machines. It can be configured to provide advanced features like high availability, fault-tolerant and recoverable storage.

- **Swift** provides *Object Storage*, which is used for storing all the additional assets and data that are required for the deployment of the virtual machines.
- **Horizon** provides a web based interface that can be used to monitor and interact with the cloud platform.
- **Heat** is the orchestration component of OpenStack. It adds templating capabilities to OpenStack by providing a template language, based on YAML, that can be used to define a topology with virtual machines, networks and every other details. The template can then be used to deploy the topology multiple times and also allows to define arguments that are used to modify certain aspects of the topology when it is deployed.
- **Celiometer** is the telemetry service that is designed for collecting data on the usage of the cloud infrastructure, which can then be used to provide customer billing if needed. This is the only service that is not useful for a Cyber Range, but could be integrated to offer paid access to third parties.

2.2.2 Networking in OpenStack

As presented before, Neutron is the OpenStack component that handles all the operations related to the networking of the virtual environments. It provides two kind of networks that can be used by the virtual machines. *Tenant* networks are fully virtualized and it is possible to create an unlimited amount of them. On the other hand, *Provider* networks are based on the physical networks and allows the virtualized environment to communicate with the real network. This is used when there is the need to connect to the Internet from a virtual machine. For both network types, it is possible to extend the virtual network between multiple physical server, which is needed when multiple compute nodes are used. More details on how both types of network are implementation is provided below.

Neutron is highly configurable to adapt to the physical infrastructure and environment, by using different plugins and agents. In particular, the mechanism used to interact with the Layer 2 of the physical network makes the most difference, with multiple option to transport frames between network and compute nodes, e.g. VLAN, VXLAN and GRE.

Open vSwitch (OVS) [14] is the most commonly deployed agent for both level 2 and level 3 management, also due to its better performances in comparison to the Linux Bridges agent [15].

OVS can be configured and adapted to the layout of the physical servers and required features. When a dedicated Network Node is used, the components and connections that OVS creates for connecting a virtual machine to a tenant network is shown in Figure 2.2

In particular, it creates:

- Tap port connected to the virtual machine.
- Linux Bridge that acts as local switch for the tenant network. Each virtual machine deployed on the same Compute Node that is connected to the same tenant network, will have its tap port connected to this bridge.
- OVS Integration Bridge integrates the networking services provided by OVS, such as DHCP and routing between networks, and connects to the tenant network bridge described before. This bridge is deployed on every compute and network node.
- OVS Tunnel Bridge is used to extend the tenant network between multiple physical servers by tunnelling it, which can be done with different technologies depending on the OVS configuration. Usually VLAN or VXLAN technologies are used. Therefore, the OVS Tunnelling Bridge connects to the OVS Integration Bridge and then forwards the tunnelled packets to the destination node

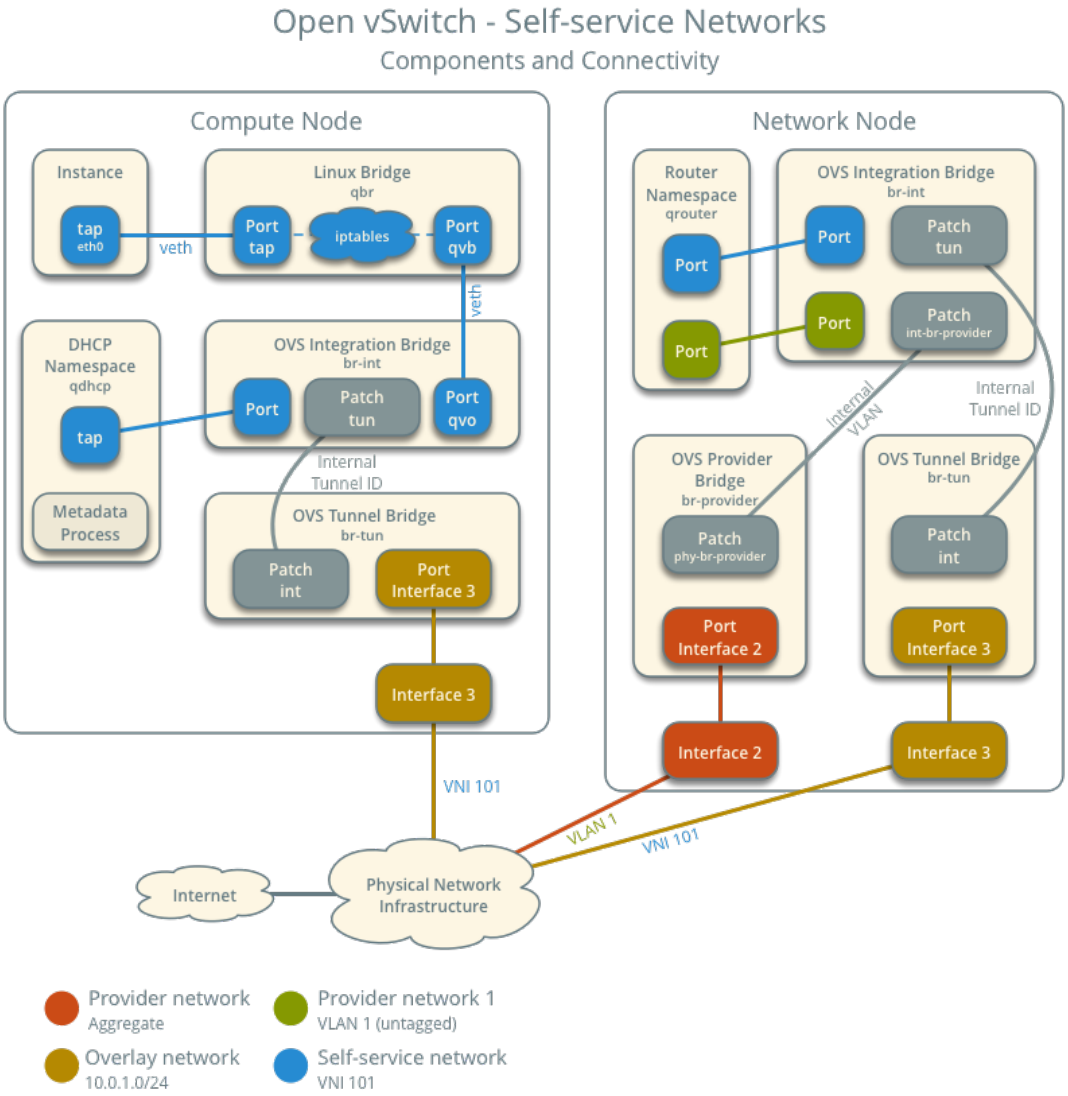

Figure 2.2: Neutron - OVS components structure

on the Overlay network. It is possible to configure OVS such that it uses a dedicated physical network for the tunnelled traffic. When OpenStack is deployed on a single physical node the OVS Tunnel Bridge is not used since all the components are installed locally.

- On the dedicated network node, the configuration is similar, with the addition of the routing components and the OVS Provider Bridge, which connects the provider networks to the physical network infrastructure.

As presented before, OpenStack capabilities can be extended by deploying multiple compute nodes. Therefore, it is possible that virtual machines connected to the same tenant network are deployed on different compute nodes. In the case that this happens, Neutron needs to route the tenant network traffic from one node to the other. An example of the traffic flow from Instance 1 to Instance 2 is shown in Figure 2.3, that shows the traffic going through the following components:

1. Node 1 - Virtual machine 1 tap port

2-3-4. Node 1 - Tenant network Linux Bridge

5-6. Node 1 - OVS Integration Bridge

7-8 Node 1 - OVS Tunnel Bridge, VXLAN encapsulation

9-10-11. Physical Network, from Compute Node 1 to Compute Node 2

12-13. Node 2 - OVS Tunnel Bridge, VXLAN decapsulation

14-15. Node 2 - OVS Integration Bridge

16-17. Node 2 - Tenant network Linux Bridge

18-19. Node 2 - Virtual machine 2 tap port

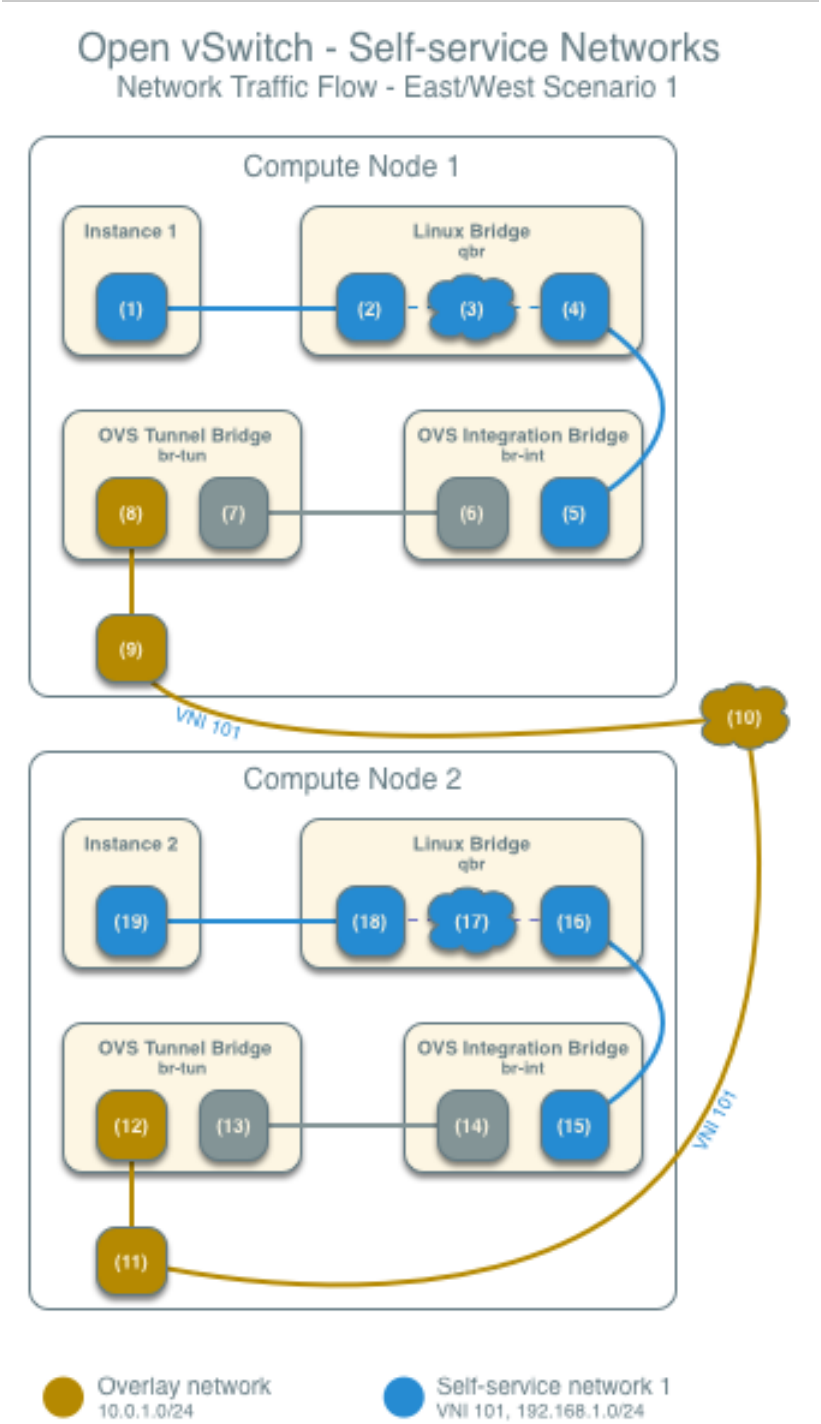

Figure 2.3: Neutron - OVS flow between multiple compute nodes

2.2.3 Limitations of OpenStack

As described before, OpenStack is composed of many components and is highly configurable, which makes it flexible and easy to adapt to different data-center layouts but also increase the complexity of the setup phase. However, the installation is usually done only once, so it is not an aspect that introduce overhead during the utilization of the Cyber Range.

Instead, one missing feature that is more important is the lack of virtual Layer 2 cable entity. When specifying a network interface for a virtual machine, it is required that it is assigned an IP address and thus that it is connected to a Layer 3 network. This is usually not a limitation, but if we want to simulate two machines that are directly connected with a Ethernet cable, we have to create a dedicated virtual network and assign IP addresses to both virtual machines' interfaces.

The second aspect that can be considered a limitation for certain uses cases is related to HEAT, the orchestration component, and in particular the templating language it uses, HOT. HOT can be used to describe every element of the topology and all the details of virtual machines, networks and other element of the virtual environment. While this is useful for normal use-cases, it results in templates unnecessary verbose when OpenStack is used as base infrastructure of a Cyber Range, because for this use-case it is required to specify some details that are not necessary. Section 3.3 provides more detail on HOT and shows an example of its verbosity.

3 Existing solutions

This Chapter presents existing Cyber Ranges and templating solutions. First, various implementations of Cyber Ranges and their use-cases are presented, then Section 3.2 focuses on the templating capabilities of some of the analysed Cyber Ranges and lastly Section 3.3 presents HOT, the template language used by the orchestration component of OpenStack, HEAT.

3.1 Existing Cyber Ranges

ViSR [16] was developed to provide a virtualized environment that is realistic, repeatable and reproducible, that can be used to execute valuable experiments. It uses OpenStack as a base infrastructure with the addition of custom modules developed for the specific use case. In particular, the work was focused on analyzing and improving the realism of the virtual environment. For this, they developed Haystack [17], a tool that can emulate user interactions with the virtual environment. Heat is used for defining and deploying network topologies, but an additional feature was developed to allow users to create a HOT template from an already deployed topology.

The ***Cyber Range Automated Construction Kit (CRACK)*** is a framework developed to help with all the phases required to develop a scenario, including the design, automated verification, deployment and testing [18]. For this, they defined a Scenario Definition Language, called CRACK SDL as an extension of TOSCA [19],

which uses the YAML format. CRACK SDL is similar to HOT, the language used by HEAT, the orchestrator component of OpenStack, with additional elements used to define characteristics of the scenario other than the network topology. It also supports the derivation of elements' types, increasing the extensibility of the language. Moreover, in addition to the simple topology, CRACK SDL supports the definitions of Vulnerabilities, Users, Principals, Software, Artifacts and Policies, enabling the scenario designers to automatically deploy on the cyber range also all the required software and (mis)configurations. In order to verify the correctness of the scenario, CRACK uses logic programming, in particular Datalog [20], converting the SDL definition to a Datalog specification and performing formal verification to check that the specified scenario and its objectives are correct and coherent.

KYPO CRP is the "Cyber Range Platform" developed by the Masaryk University [21]. They use a topology definition based on YAML, which specifies hosts, routers, networks, router mappings and network mappings, which define the connections of routers and hosts to networks respectively, and groups of hosts or routers. A second group of YAML files, is used for defining the software configurations required, which are then applied during the deployment to the hosts defined in the topology file using Ansible. The last scenario configuration component is a JSON file that contains the definitions and properties of the training.

The ***Doman Specific Language (DSL)*** proposed for the Norwegian Cyber Range (later Norway 1) uses a YAML-based language to represent the scenario [8]. The DSL definition can also be generated using a graphical tool, which they call "cyber security strategy game". The DSL can be used to define multiple aspects of the scenario, including the properties of the scenario, nodes, which are used to define virtual machines and their configurations, routers, services, vulnerabilities, challenges, teams, agents, phases, objectives and rules. For the deployment, they provide a compiler for DSL that performs Syntax and semantic validation of multiple

aspects of the scenario. The result of the compilation are three artifacts: the HEAT template to be deployed on OpenStack, an Ansible template, that is used to deploy software configurations on the virtual machines, and an external artifact containing additional information.

The Norwegian Cyber Range developed also a different and more complex system (later Norway 2) for automating and making the creation of scenarios efficient [22]. This system uses a JSON-based language for generating the topology, injecting vulnerabilities, defining attacker and defender behavior and providing traffic generation. The system uses a formal verification mechanism that translate the JSON language to a Datalog formal model, similarly to what is done in CRACK. If the formal verification succeeds, then the language is used to generate the infrastructure orchestration artifacts: a HEAT template for the topology deployment, the vulnerability injector scripts, which use SSH to connect to the virtual machine and perform the required configurations, attacker agent scripts, defender agent artifacts, that instruct the defender agent on how to response to certain attacks, and the traffic generation scripts. The artifacts are then used by the scenario orchestrator to execute the scenario.

The framework developed in [23] uses the ***Virtual Scenario Description Language (VSDL)*** domain-specific language for defining the topology and high-level details of the scenario. The language uses a custom syntax to describe the elements of the scenario and it is compiled to a satisfiability modulo theory (SMT) specification which is then submitted to the CVC4 [24] solver. If the specification is satisfiable, the solver returns a model, that is translated to Terraform and Packer scripts that are finally used to deploy the scenario on OpenStack.

CRATE is the Cyber Range developed by the Swedish Defence Research Agency (FOI) [9]. It is composed of multiple components that allow to create emulated environments but also include hardware devices. The different components communi-

cate through the "Core Api" and the CRATE Exercise Control (CEC) [25] is used to manage the Cyber Range, by deploying and monitoring the virtual topologies. The system allows to automatically provision the virtual environment, including specific configuration of the virtual machines, and provides features such as user actions emulation, data collection and the injection and execution of different tests. This Cyber Range was developed to provide a system capable of performing controlled experiments as well as trainings and exercises.

CytrONE [26] is a Cyber Range used for trainings that focus attacks reproducing known vulnerabilities, forensic analysis and defense skills. Therefore, the design of this system focus mainly on trainings and they provide also ***CyPROM*** [27], a scenario progression management module that guides the user through the training executing the required steps in the virtual environment based on the single user progress. Moreover, the Cyber Range in integrated also with the Moodle LMS to provide a a better learning platform.

CyberVan [28] is a Cyber Range designed to provide a high fidelity simulation of mobile wireless tactical networks and strategic networks. It support multiple virtualization technologies for the virtual hosts and includes various elements to improve the realism of the testbed, such as traffic simulation, including malicious traffic, cyber attacks mechanisms and realistic network topologies.

INSALATA [29] is a framework that focus on the automatic replication of real network in virtualized environments. In order to achieve this, the management component receives data from the collector and gives instructions to the deployment component to modify the virtual environment. The collector unit, uses multiple techniques to obtain information on the network, both active, such as `Nmap`, `SSH` and `SNMP`, and passive, like `tcpdump`. Moreover, it is possible to manually provide some information on the physical topology with XML files. The deployment component uses the `XEN` virtualization platform to deploy the virtual copy of the network.

The analysis provided in [30] shows a Cyber Range implementation focused on a very specific use case, the cyber-physical tests for *Autonomous vehicle shuttles*. In particular this is used for testing possible attacks that target the autonomous navigation algorithms. In this case, the use of real hardware becomes more important as full software simulation might not provide high enough fidelity. However, the use of hardware reduces the flexibility and ease of use of the testbed. Anyway, this is a good example of use case specific Cyber Range, but this solution will not be considered later because of the very limited and unique scope that differs from the more generic Cyber Range that we aim to develop.

3.2 Templating capabilities of existing Cyber Ranges

We are particularly interested in systems that can be used to deploy various kind of virtual topologies and that can be applied to multiple use-cases. Therefore, of the cyber ranges presented before, we selected those that provide a clear and detailed description of the language used.

CRACK uses a YAML syntax which is almost identical to HOT, so it is possible to generate all the topologies supported by OpenStack. However, this means that it also derives the verbosity and complexity issues of HOT, as shown in the next section. One important feature of their implementation is the formal verification, which makes sure that the topology and all the additional software configuration are correct. Moreover, for the configuration of the hosts, many details can be modelled directly in the language. Finally, the language supports the derivation of element types, providing the option for the users to extend the language.

Kypo uses also a YAML file for describing the topology but it is much more compact than HOT, since it is structured differently and it does not require to specify some details like ports elements or IP addresses allocation ranges. However, it provides only three elements for constructing the topology: routers, networks and host.

From the language description it seems that the routers are always interconnected, but it should be possible to not connect a network to a router, thus the topologies that can be created should be quite flexible. Moreover, they use separated Ansible files to provide software configurations to the hosts.

Norway 1 uses a YAML configuration file as the two previous cyber ranges. They provide only two elements for representing the topology: Node, which represent a virtual machine, and Router. The Router is used to define the subnets that are connected to it and in the Node definition it is possible to define a list of router and subnet pairs to which connect the Node. Since all the routers are connected together, the structure of the topologies that can be represented is limited. For the software configuration, the language provides elements such as Service, Vulnerability and Challenge to define the configuration of the hosts. These elements are then used to generate Ansible templates which are used during the topology deployment. Finally, for the validation they use both syntax and semantic validation.

Norway 2 is the only cyber range to use a template language based on JSON and it uses Datalog to execute a formal verification of the scenario as in CRACK. The language provides only two elements for the topology definition: Machine and Subnet. The Subnet require the ID of a Network already existing in OpenStack, thus it seems that a network must be first manually generated. Moreover, the grammar definition seems to suggest that a Machine can be dependent, which is how they define the connection relationship between Machine and Subnet, only on a single subnet. This really limits the topology structure, or requires then some manual intervention directly on the OpenStack infrastructure. Moreover, they provide an example of the creation of a large topology, with similar or identical networks replicated multiple times. For this example, they generated the template for a single network and deployed it multiple times, which probably required a final manual intervention to interconnect the networks. For the software configuration, the lan-

guage supports the definition of vulnerabilities, that are then used to deploy the required software in the Machines using SSH. Moreover, the language supports also the modelling of attacker and defender behaviors, as well as traffic generation, which are used to emulate the actions of red and blue teams. This aspect is especially useful in the *dry run* phase, where the scenario is executed to verify that everything works correctly.

VSDL uses a custom syntax that defines only two type of elements for describing the topology, node and network, and it allows to define properties of the elements based on logical connectives instead of fixed ones, e.g. `disk is larger than 100 GB`. Moreover, it includes time-based modifiers to enable or disable properties at specific moments of the execution, which is a unique feature not present in the other analyzed works. For the software configuration, the language provides two properties in the node element that can be used for defining the software and vulnerabilities that should be present in the virtual machine. This configuration is then handled by Packer and there seems to be no option to define and use custom scripts directly in the template. Finally, there is no feature to produce large topologies easily, which is an aspect recognized also by the authors.

Table 3.1 provides a summary of the features included in the various templating solutions.

3.3 HOT Language

As seen in Chapter 2 *Heat* is the component of OpenStack that enables the use of templates to allocate topologies of virtual servers, networks and other resources. HOT, *Heat Orchestration Template*, is the templating language that is used by Heat. Moreover, as shown in the previous section, OpenStack is used as a base infrastructure by multiple Cyber Ranges, but most of them prefer to use a different language that is then translated to HOT and finally used to deploy the virtual

Cyber Range	Language format	Topology elements	Verification	Host configuration
CRACK	YAML	Similar to HOT: Host, Network, Subnet, Port	Formal verification with Datalog	Language elements to model Software, Vulnerabilities, Users and more.
Kypo	YAML	Host, Router, Network and network, router mappings	Not specified	Ansible YAML configuration files
Norway 1	YAML	Node and Router. Subnets are defined by routers	Syntax validation and semantic validation	Language elements to model Software, Vulnerability, Challenge, Teams and more
Norway 2	JSON	Subnet, Machine	Formal verification with Datalog	Language elements to model Services, Vulnerabilities, Challenge
VSDL	Custom grammar	Network and Node	Formal verification with CVC4	Software and vulnerabilities properties

Table 3.1: COMPARISON SUMMARY

environment. This approach is very common because of different requirements for the templating feature of the Cyber Ranges, such as including additional details of the scenario, and also due to the verbosity and complexity of HOT. We now present it a bit more in details and show an example of its verbosity.

HOT is based on YAML and is composed of 5 sections used to define different aspects of the virtual environment [31]. The main section is the `resources` one, where all the components elements that compose the virtual topology are defined with all their parameters and details. The other sections are used mainly to provide some customization of the template, such as defining parameters that can be defined at deployment time to modify certain aspects of the topology.

An example of a very simple topology is shown in Listing 3.1. The topology contains only a virtual machine *VM_1* that is connected to a virtual network *net_1* with the fixed IPv4 address of `1.2.3.4`. In order to achieve this simple topology, with hot it is required to define the following entities in the `resource` section:

- The virtual network *net_1*.
- A sub network *subnet_1*, related to *net_1*, which is used to specify the IPv4 allocation range of the network.
- A port, *VM_1_port*, which defines a connection on the network *net_1*, with a fixed IPv4 address.
- The virtual machine *VM_1*, which uses the port *VM_1_port* to connect to the network *net_1*.

Listing 3.1: Example of the HOT language

```
heat_template_version: 2016-10-14

description: Simple HOT example

resources:
  net_1:
    type: OS::Neutron::Net
    properties:
      name: net_1

  subnet_1:
    type: OS::Neutron::Subnet
    properties:
      name: subnet_1
      network_id: { get_resource: net_1 }
      cidr: "1.2.3.0/24"
      gateway_ip: "1.2.3.1"
      allocation_pools:
       - { "start": "1.2.3.1", "end": "1.2.3.254" }

  VM_1_port:
    type: OS::Neutron::Port
    properties:
      network: { get_resource: net_1 }
      fixed_ips:
        - "ip_address": "1.2.3.4"

  VM_1:
    type: OS::Nova::Server
    properties:
      image: image_name
      flavor: flavor_1
      networks:
        - port: { get_resource: VM_1_port }
```

HOT allows to define many additional details for every type of resource composing the topology, other than those shown in the example. However, when OpenStack is used as the base infrastructure for a Cyber Range, in most use-cases many of the additional details are not required. For example, the port entity is a concept that it is usually not required nor used in a Cyber Range, instead a simpler mechanism to define the connection of a virtual machine to a network is desired. The same applies to the sub-network entity, where its properties could be defined directly in the network entity.

The possibility to specify all these details make HOT a very complex and powerful language, but at the same time very verbose and not ideal for many Cyber

Ranges application, given the added complexity that is not required.

Moreover, when there is the need to define topologies that are composed by a high number of similar entities HOT provides limited features to avoid repeating multiple definitions of the similar resources.

Therefore, these aspects limit the usability of HOT as a direct templating system for Cyber Ranges, but it can still be used as a target for the translation of other systems, which is a solution adopted by multiple Cyber Ranges, as shown in the previous section.

4 Specification and Design

This Section presents the design process for out Cyber Range and in particular the new Description Language that we use for the templating system. First the general requirements of the Cyber Range are provided with all the elements it must be able to simulate. Then these requirements are used to create some base topologies that should be easily recreated. From the topologies we extract the actual features required for the Cyber Range and the Description Language. Finally, the language grammar is presented.

4.1 General specifications and requirements

For the design of the Cyber Range structure and features we started with defining the specific use cases that it should support.

In particular, we are interested in performing security testing of different networking components, but we want to be able to use the Cyber Range also for trainings and demonstrations. Therefore, a flexible Cyber Range that can support many different topology elements and configurations is required.

The general features that our Cyber Range infrastructure should support are the following:

- **Automatic deploy** of virtual machines, networks and other components of the topology. The Cyber Range should interface with the chosen virtualization

technology and automatically setup the environment and start the virtual machines.

- **Scalability**. It should be easy to extend the Cyber Range in order to support the virtualization of topologies with a higher number of elements.

- **Isolation**. When testing certain security features we want to have a virtual environment that is completely isolated from the real external network. Therefore the Cyber Range needs to have private virtual networks. Moreover, when multiple experiments are executed concurrently on the same Cyber Range infrastructure, there is the need to keep the different virtual environment isolated.

- **Initialization of virtual machines**. It is often required to provide some initial configuration or data to the virtual machines, such as additional network settings or commands to execute after the booting sequence is completed.

- **Templating system**. The Cyber Range needs to support the creation of topologies from a template file. Ideally, the templating system should be able to easily model various aspects of the virtual environments and provide features to facilitate describing very large topologies.

- **Graphical interface**. A nice feature to have for a Cyber Range is a graphical interface that allows to see and interact with the deployed topologies.

- **Access to the virtual machines**. It is needed to access the virtual machines to execute the tests, so the Cyber Range must provide a mechanism to achieve that.

Regarding the security testing, we identified the following fundamental configurations and features of network security components that we are interested in testing:

- **Base components**, such as virtual machines, Firewalls, Intrusion Detection Systems (later IDS) or Intrusion Prevention Systems (later IPS).
- **Multi-link** connections, where two machines can correctly communicate using two different connection paths. This simulates a network that has multiple ISP connections for improved reliability.
- **VPN Multi-link** connections, similar as the one described before, but in addition the two nodes should use VPN connections.
- **Dynamic routing** that similarly to Multi-link occurs when multiple paths are available for delivering packets.
- **Firewall/IDS clustering**, differently from single Firewalls a cluster can provide higher reliability.
- **Layer 2 Firewall/IDS** to not limit the testing to only Layer 3 systems.
- **Performance** of security systems under heavy load.

4.2 Topologies

Based on the requirements identified before, we defined four base topologies that can be used to test them. The Cyber Range should provide enough features to easily deploy the topologies and a templating system to easily describe them.

4.2.1 Topology 1

The first topology, shown in Figure 4.1, includes two set of networks, both behind a Firewall, that are divided by some external networks. In particular, each Firewall is connected to two external networks and it has three internal interfaces for three different sub-networks:

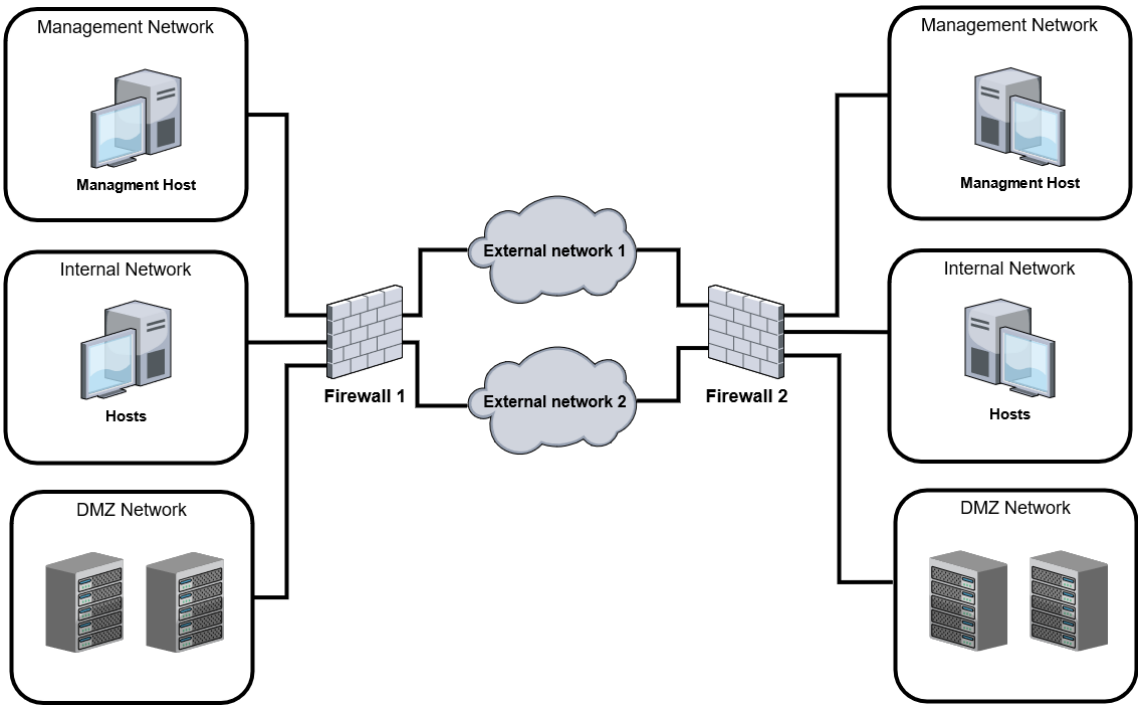

Figure 4.1: Topology 1

- Management, which includes the host used to configure the Firewalls.
- Intra, the internal network used by normal hosts.
- DMZ, where servers that expose services accessible from the external networks are located.

We can use this first topology to test multiple configurations, such as *multi-link*, since the two private networks are connected to two different external networks via their Firewalls. The same applies for the VPN multi-link and dynamic routing requirement. Moreover, many base components are already tested with this topology, with only IDS and IPS missing.

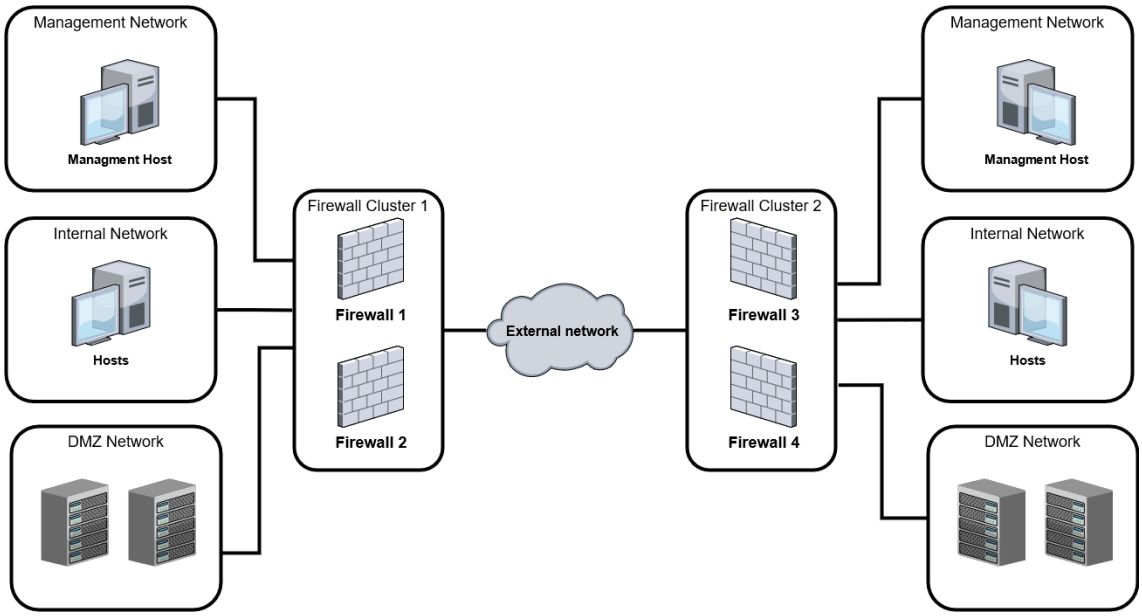

Figure 4.2: Topology 2

4.2.2 Topology 2

The second topology, shown in Figure 4.2, is very similar to the first topology, but instead of single Firewalls, a cluster composed of two Firewalls is used for separating both private networks from the external ones.

This topology is able to test all the configurations of the first topology, but focuses on the clustering of Firewalls, which is a feature tested only in this topology.

4.2.3 Topology 3

The third topology, shown in Figure 4.3, is quite different from the previous two. This topology includes a single external network and many, 8 or more, private networks, each protected by a Firewall. One of the Firewalls protects also the management sub-network, as seen in the previous topologies.

This topology focuses on testing the performance of the security systems when a high number of nodes, and thus a heavy load, is used.

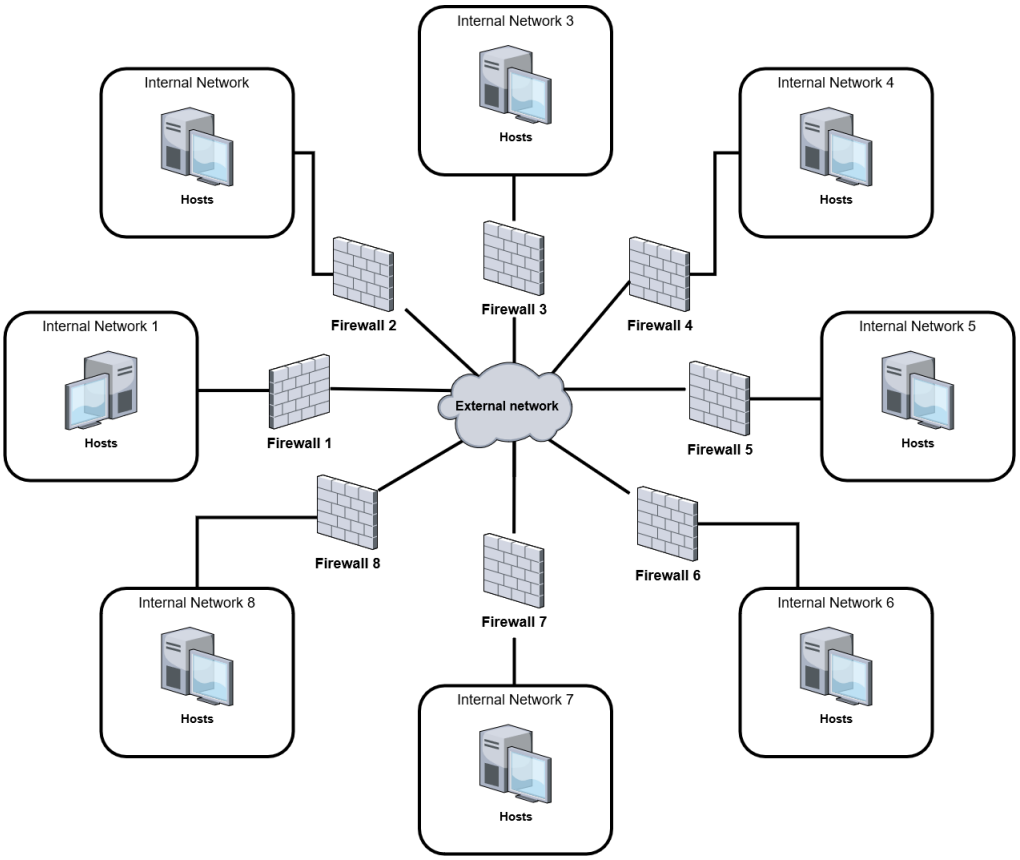

Figure 4.3: Topology 3

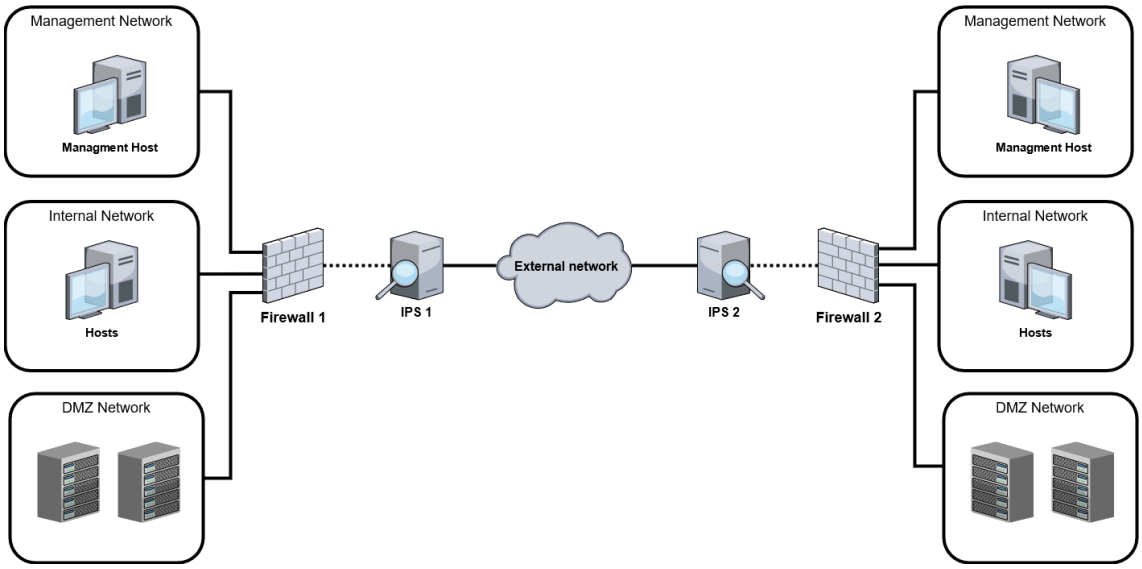

Figure 4.4: Topology 4

4.2.4 Topology 4

The last topology, represented in Figure 4.4, is similar to the first one presented, but only one external network is used and both private networks have an additional IPS, which could also be replaced by a IDS, that is connected between the Firewalls and the external network.

Layer 2 IPS or IDS are uniquely used in this topology, which is the main characteristic tested with it. This requires the use of a Layer 2 connection, that is not used in any other topology.

4.3 Cyber Range infrastructure

The general requirements for the Cyber Range infrastructure identified in Section 4.1 are supported by various features and components of OpenStack. As seen in Chapter 3, OpenStack is a popular platform that is adopted to create the base infrastructure of many Cyber Ranges.

In particular, the requirements are fulfilled by the following features:

- Nova provides automatic deploy of virtual machines, their configurations and initialization. It supports different virtualization mechanisms. Moreover, it is possible to deploy Nova on multiple physical servers, making possible to extend the virtualization capabilities of the Cyber Range.

- As shown in Section 2.2.2, Neutron supports tenant networks which are not related to real networks, allowing to create virtual networks that are completely isolated.

- OpenStack has a graphical web interface to interact with the virtual environments, provided by the Horizon component, in addition to the command lines utilities.

- It is possible to access and interact with the virtual machines using different protocols, such as VNC, Spice, RDP, and MKS.

- The templating system of OpenStack is provided by HEAT but the language it uses, HOT, is not ideal for a Cyber Range, as shown in Section 3.3. However, we can use a different language and then translate it to HOT, which is done in many other Cyber Ranges.

This shows that OpenStack is a good base infrastructure and that we can use it to develop our Cyber Range. Moreover, it is released as open-source software, which makes it possible to make changes and adapt it to our need if we require aspects that are not currently supported. For example, the four testing topologies shows that, in addition to the requirements identified in Section 4.1, we require the use of Layer 2 connections between two virtual machines and a virtual machine and a network. This type of connection is not currently supported by Neutron and Chapter 5 shows also how this feature was integrated.

Regarding the templating system for our Cyber Range, we explained already that HOT is not suitable. Therefore we decided to develop a new language, later called Description Language, that is used to describe the topologies and it is then translated to HOT in order to use HEAT to automatically deploy the scenarios.

4.4 Description Language

4.4.1 Features required

From the four topologies shown before, we defined the following features that the Description Language must support, in order to be able to recreate them:

- **Hosts** entities: represent the virtual machines that will be deployed. For every host we should be able to specify some options, such as the amount of resources dedicated to the machine, usually called flavor, the image that is used to boot the machine and some initialization option. Moreover, being able to specify special hosts, like firewalls, should be supported.

- **Networks**: the virtual networks that the hosts will be connected to. The language must support certain options also for the network entities, such as the CIDR used.

- **Network connections**: each host entity can be connected to one or more network, so the language needs to provide a function to specify the connections between hosts and networks.

- **Layer 2 cables**: for the fourth topology, described in Section 4.2.4, it is required to connect two hosts directly, without the need of defining a network, since the two devices will communicate directly and they will not require IPv4 addresses assigned on the interfaces in order to properly work. In particular,

in the fourth topology this is used when the Layer 2 IPS or IDS is placed between the Firewall and the external networks. The Firewall will not see that its interface is not connected directly to the external network, since the Layer 2 IPS or IDS operate in a transparent way. Therefore, defining a network between the two is unnecessary.

- **External networks**: this entity represents a network that we do not need to manage directly, for example it could represent the ISP network. It could be possible to use the normal Network entity, but it might be useful to a separate one to reduce the amount of detail that is required, or also to provide a different semantic value that can be used to better document the topologies.

- **Clusters of hosts**: It is possible to manually create a cluster of hosts by just using the previously described entities, but the Description Language should provide a simpler mechanism to create a cluster of virtual machines, which should also be more semantically explicit, in order to obtain a more easily understandable topology description.

Moreover, the Description Language should ease the creation of a large number of similar entities, in order to reduce the verbosity of the topology description.

Finally, it would be interesting to have some features that improve the extensibility to the language. This can be useful for both the creation of large topologies and the flexibility and customization that might be required in certain scenarios and use-cases.

4.4.2 Grammar specification

Given the requirements specified above, we created the following grammar.

The Description Language is composed by a list of statements. Each statement represents and describes an entity, is terminated by the ; character and multiple

```
<type> <name> [<attribute>(<value1>, ...)]* ;
```

Figure 4.5: Statement syntax

white-spaces are ignored.

Each statement is constructed with the syntax shown in Figure 4.5, which is composed by the type of the entity, its name, which must be unique in the template, and multiple attributes, when needed.

The type of the entity and its name are always required. Then multiple attributes can be used, depending on the type of entity and each attribute can have multiple values.

The format of entity's names and attributes' values can be quoted strings or identifier format `[a-z,A-Z,0-9][a-z,A-Z,0-9,_]*`, with some exception such as IPv4 addresses or CIDR values.

Listing 4.1 shows an example topology described with the Description Language grammar.

Listing 4.1: Example of the Description Language grammar

```
ext_net public;

host attacker
    image(attacker_img)
    interface(public);

network intranet cidr(192.168.10.0/24);
network dmz cidr(192.168.20.0/24);

cable firewall_to_ips;
cable "ips to public" connection(public);

host ips image(ips_img)
    interface("ips to public")
    interface(firewall_to_ips);

firewall FW
    interface(firewall_to_ips)
    interface(intranet, gateway)
    interface(dmz, gateway)
    cinit("fw_setup_script.sh");

host simple_host
    interface(intranet);

host server1
    image(server_img)
    interface(dmz, 192.168.20.1)
    cinit("server_setup_script_1.sh");

host server2
    image(server_img)
    interface(dmz, 192.168.20.2)
    cinit("server_setup_script_2.sh");

cluster server_cluster
    host(server1)
    host(server2)
    interface("dmz", 192.168.20.10);
```

The following entity types are currently supported:

Network

Create a virtual network with the given name and cidr. The attribute `cidr` is mandatory and defines the range of IPv4 addresses that will be used in the network.

```
network <name> cidr(<cidr value>)
```

Figure 4.6: Network entity syntax

Host

Create a virtual host.

```
host <name> [image(<image_name>)]?  [flavor(<flavor_name>)]?
    [interface(<network name> [, <ip> | "gateway"]?  )]+
        [cinit(<init_file> [, (<from>, <to>)]* )]?
```

Figure 4.7: Host entity syntax

The syntax of the host entity is more complicated because there are more attributes:

- **image**: Select the image that will be used during the boot of the host. If not specified, the default image will be used.
- **flavor**: Select the flavor to use for the host. If not specified, the default flavor will be used.
- **interface**: Connects the host to the given network, external network or cable. Each host needs at least one interface. The first value of the interface attribute is the name of the network, external network or cable. The second one is optional and is used to specify the IPv4 address that the host will have. The value can be an IPv4 address or the keyword `gateway`, if the host will be the default gateway for the network. If the second value is not specified, the host will receive a random IPv4 in the network range from the DHCP service.
- **init**: Select the file that cloud-init should use for the host initialization. If not specified, no initialization file will be used. It is possible to specify a list of pairs of strings that will be replaced in the given file.

External network

Create an virtual network that represents an external network that we do not want to manage, such as a ISP network. It is not very different from the normal network entity, but it is useful to make the topology semantically more clear.

```
ext_net <name> [cidr(<cidr>)]?  [route(<ext_net name>)]*
```

Figure 4.8: External network entity syntax

The cidr attribute is the same as the network entity one, but it is not mandatory. The route attribute is not mandatory, can be repeated multiple times and represents a connection to another external network, so that traffic can be routed between the two. It does not matter on which external network the attribute is specified.

Cluster

Create a cluster of hosts. A cluster could be created also manually, but the cluster entity makes it less verbose and more semantically understandable.

```
cluster <name> [cidr(<cidr>)]?  [ [firewall|host](<name> [,
<ip_addr>]?)]+ [interface(<network name> [, <ip> | "gateway"]?  )]+
```

Figure 4.9: Cluster entity syntax

The cidr attribute is used to define the cidr of the management network that is created for the cluster. Every host is connected also to this network when added to a cluster. If it is not defined, a default value will be used.

Multiple host attributes can be defined and the keyword firewall can also be used, since in the test topologies the cluster is used mainly for firewall instances. For every hosts, the name of an existing instance must be declared and optionally it is possible to specify the IPv4 address that the host will have in the management network.

Multiple interface attributes can be used, similarly to the host entity, and it is used to specify a connection between the cluster and the network with the given name. Every hosts must be connected to all the networks on which the cluster has an interface. The IPv4 address of a cluster interface, if specified, will be assigned to all the interfaces, on the same network, of all the host or firewalls in the cluster, as virtual IPv4 addresses. It is possible to use the keyword `gateway` if the cluster will be the default gateway for the network. If not specified, the cluster will get a random IPv4 address in the network range, similarly to a host instance.

Cable

Create a virtual Layer 2 connection between two hosts or a host and a network. Connecting two networks using a cable is currently not supported.

```
cable <name> [connection(<entity_name>)]?
      [connection(<entity_name>)]?
```

Figure 4.10: Cable entity syntax

The attribute connection can be used two times or less. It is possible to connect a host to a cable also using the interface attribute in the host instance. A cable must always be connected to two different entities, hosts or networks, independently from how the connections are specified.

Firewall

The firewall type is derived from the host type, which means that it is equivalent to a host type but it uses different default values. In particular, the default image and flavor are different from the host ones. Therefore, the syntax to create a firewall is the same as the host one, but with a different type name.

```
firewall <name> [image(<image_name>)]?  [flavor(<flavor_name>)]?
      [interface(<network name> [, <ip> | "gateway"]?  )]+
        [<init_type>(<init_file> [, (<from>, <to>)]* )]?
```

Figure 4.11: Firewall entity syntax

All the attributes of the firewall type are defined equally to the host ones. Therefore, it is possible to directly use the host type if values different from the default ones are used for the image and flavor attributes. However, it is more semantically significant to a reader if the correct type is used.

4.4.3 Grammar specification - Preprocessor

In addition to the grammar specified in the previous section, additional features are needed to improve the extensibility and reduce the verbosity of the topology descriptions. The structure of these additional constructs and their grammar is different from the statements syntax described before and it is interpreted in a different phase, called preprocessor which is executed earlier than the parsing of the statements. This allows to achieve scalability in a more flexible way.

Listing 4.2 shows the same topology described in Listing 4.1 integrated with the preprocessor directives.

This grammar is composed by the following **directives**:

Listing 4.2: Example of the Description Language grammar and preprocessor features

```
// Topology that showcase all the features
// of the language

ext_net public;

host attacker
    image(attacker_img)
    interface(public);

network intranet cidr(192.168.10.0/24);
network dmz cidr(192.168.20.0/24);

cable firewall_to_ips;
cable "ips to public" connection(public);

host ips image(ips_img)
    interface("ips to public")
    interface(firewall_to_ips);

firewall FW
    interface(firewall_to_ips)
    interface(intranet, gateway)
    interface(dmz, gateway)
    cinit("fw_setup_script.sh");

// Host with default image
host simple_host
    interface(intranet);

#for i (1,3):
host server{i}
    image(server_img)
    interface(dmz, 192.168.20.1{i})
    #if ({i} = 1):
    cinit("server_setup_script_1.sh");
    #else
    cinit("generic_setup.sh", (ID, {i}));
    #endif
#endfor

cluster server_cluster
    #for i (1,3):
    host(server{i})
    #endfor
    interface("dmz", 192.168.20.10);
```

For directive

The For directive is composed by a header, Figure 4.12, and a footer, Figure 4.13. Every character between the header and the footer is considered part of the body.

```
#for <label_name> (<range_start>, <range_end>):
```

Figure 4.12: For directive: header syntax

```
#endfor
```

Figure 4.13: For directive: footer syntax

The header specifies a label that can be use in the body with the syntax `{<label_name>}` and a range of values, with both ends inclusive. For each value in the range, the body will be replicated, replacing every instance of the label, as specified before, with the correct value.

If directive

The If directive is composed by a header, Figure 4.14, and a footer, Figure 4.15. Every character between the header and the footer is part of the body and optionally, the body can be divided by the else header, Figure 4.16. If the else header is used, every character between the else header and the footer is part of the *else body*.

```
#if (<string_1> <comparison> <string_2):
```

Figure 4.14: If directive: header syntax

```
#endif
```

Figure 4.15: If directive: footer syntax

```
#else
```

Figure 4.16: If directive: else syntax

The header specifies a comparison between two strings. The only comparison supported by the Description Language at the moment is the equal comparison.

If the comparison is true, then the body is left unchanged. Otherwise, the body will be deleted. In the case that the else body is specified, using the else header, it will be left unchanged if the comparison is false, or deleted otherwise.

Nesting

It is possible to nest directives, in order to create more complex structures and obtain a more powerful extensibility of the Description Language, that allows to describe large topologies more easily and less verbosely.

It is possible to use a label declared in a for directive in the comparison of a if directive, or in the body of the if directive, if the second directive is contained in the body of the first one.

The only limitation to the nesting of directives is that it is required to properly nest directives. Therefore, it is not possible to have the footer of the outer directive before the footer of the inner directive. The Listing 4.3 shows the described incorrect nesting.

Listing 4.3: Example of incorrect nesting of directives

```
1 #for i (1, 3):
2     #if ({i} = 2)
3 #endfor
4     #endif
```

Comments

The last feature supported by the Description Language is comments, which are not directly useful for the description of the topology but act as documentation.

The Description Language supports C-style single line comments, which means that when the tag `//` is found, everything until the end of the line is considered comment and ignored by the parser.

5 Implementation

This chapter focuses on the implementation of the components we developed for the Cyber Range. In particular, Section 5.1 describes the implementation of the Description Language compiler and Section 5.2 shows the other components developed for the Cyber Range infrastructure.

5.1 Description Language compiler

In Chapter 4, the design of the grammar for the Description Language was presented. It was also shown how OpenStack provides a good base infrastructure for the Cyber Range and how HOT is too verbose but can be used as a translation target for our Description Language.

Therefore, the workflow needed to deploy a certain topology will be to write a template describing the topology using the Description Language, then use the compiler to translate it to an equivalent HOT template and finally use the output to deploy the topology on the Cyber Range by either uploading it on the web interface or using the command line utilities.

This section presents the implementation of the compiler that translates between the two languages taking as input a template written with the Description Language and outputting the corresponding HOT template.

The compiler is implemented in Rust, extensively using the parsing library `NOM` [32], which allows to produce fast and correct parsers, and provides functions,

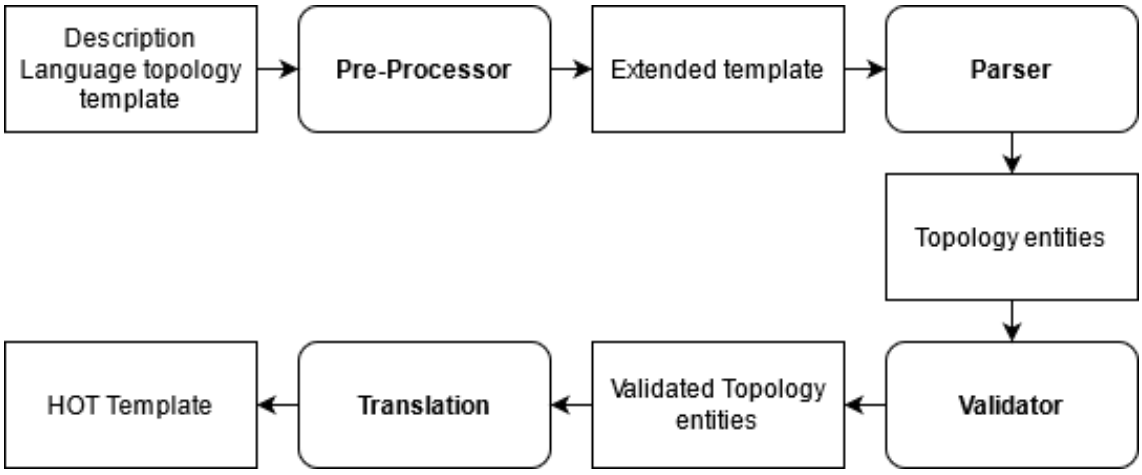

Figure 5.1: Compilation process

macros and traits to facilitate the implementation of parsers and error handling.

The only other dependency used for the compiler is the library `lazystatic` [33], which is used for creating a static configuration structure that is used in different parts of the parser and its values can be modified by setting the correct environment variables, as described later.

Compilation is divided in 4 phases, as shown in Figure 5.1, each handled by a different component:

- The *Preprocessor* is the component that parses and resolves directives. It outputs a modified version of the original file, that contains only the entities that compose the Description Language grammar, as described in Section 4.4.2.

- The *Parser* takes the result of the preprocessor as input, parses the entities and allocate all the corresponding structures that are required to represent the topology. This step verifies that the template is syntactically correct.

- The *Validator* performs various verification on the entities allocated by the parser, checking that the entities have a valid semantic and not only a valid syntax. This step also ensures that the described topology does not contain

configuration errors.

- Finally the *translation* takes all the entities that have been checked by the validator and creates the correct respective HOT template as output.

Each component is presented in more details in the following sections.

5.1.1 Preprocessor

The first step executed by the preprocessor is the removal of all the comments. To achieve this, it parses the input searching for all the instances of the tag `//` and, when found, removes every character from the start of the tag until the first end of line found.

Once the comments are removed, the preprocessor parses the resulting input again, this time creating a tree, or forest, of nested directives. In order to do so, in addition to the directives presented in Section 4.4.3 a *Base* directive is used, which is used as a container for all the characters that are not part of the other directives' headers and footers. The parsing technique used for this step works by reading character by character until `#` is found. A Base directive is created with all the characters that were parsed before the `#`. Then, the header or footer is parsed and the corresponding directive is allocated or updated. Then, the parsing algorithm starts over with the Base directive parsing.

The preprocessor's parser is constructed using a combinations of functions provided by `NOM`. Listing 5.1 and 5.2 show how the high level parsing functions for the If and For directives are constructed. The functions `if_header`, `if_body`, `else_header`, `else_body`, `if_footer`, `for_header`, `for_body` and `for_footer` are other parsing functions we defined, while the other functions used to combine and construct the parsers are provided by `NOM`.

Listing 5.1: Parsing code for If directive

```
pub fn if_cond<'a, E: ParseError<&'a str> + ContextError<&'a str>>
(input: &'a str) -> IResult<&'a str, IfDir, E>
{
    context(
        "If␣directive",
        map(
            terminated(
                tuple((
                    if_header,
                    context(
                        "body",
                        if_body
                    ),
                    opt(
                        preceded(
                            else_header,
                            else_body
                        )
                    )
                )),
                if_footer
            ),
            |(header, body, else_body)|
                IfDir {
                    header: header,
                    body: body,
                    else_body: match else_body {
                        Some(b) => Some(b),
                        None => None
                    }
                }
        )
    )(input)
}
```

Listing 5.2: Parsing code for For directive

```
pub fn for_loop<'a, E: ParseError<&'a str> + ContextError<&'a str>>
(input: &'a str) -> IResult<&'a str, ForDir, E>
{
    map(
        terminated(
            pair(
                for_header,
                for_body
            ),
            cut(for_footer)
        ),
        |(header, body)|
            ForDir {
                header: header,
                body: body
            }
    )(input)
}
```

The result of this parsing is a sequence, hence why it is a forest and not a simple tree, of nested directives. Each directive can be either:

- BaseDir: represents the test that is not part of headers and footers.
- IfDir: it is composed of a header, a footer, a body and optionally an else body. The body, and also the else body, is another forest of directives, which means that is can be composed of a single BaseDir, if there are no nested directives, or a list of nested directives otherwise.
- ForDir: it is composed of a header, a footer and a body. Its body is a forest of nested directives, as the IfDir one.

The resulting list of directives is then inserted in a stack and processed, by *resolving* the top directive in the stack iteratively. The resolve action depends on directive type:

- BaseDir: no action required. The directive is pushed in the output list.
- IfDir: the condition in the header is checked. If it is true the directives contained in the body are pushed on top of the stack. Otherwise, if the else body is present the directives contained in it are pushed on top of the stack, if it is not present, no action is needed and the directive can be discarded.
- ForDir: for every value contained in the range specified in the header, the body is replicated with a replacement of the variable label with the current iteration value in every directive that composes the body, recursively executing the `replace` function in the nested directives. Then, the resulting directives are pushed on top of the stack and the ForDir directive is discarded.

Listing 5.3: Example input of the preprocessor

```
#for i (1,3):
    network net{i} cidr(10.0.0.0/24);
    #for j (1,3):
        #if ({i} = 1):
            host "server {i} {j}" interface(net{i});
        #else
            host "vm {i} {j}" interface(net{i});
        #endif
    #endfor
#endfor
```

This process will resolve every directive and produce a list of resulting BaseDir. The list is then processed by extracting the body of each directive and concatenating it. The resulting string is the output of the preprocessor.

As an example of the preprocessor execution, Listing 5.3 shows a possible input, which is parsed and a simplified representation of the resulting forest of directives is shown in Listing 5.4. Then, the directives in the forest are resolved and the output produced by the parser is shown in Listing 5.5.

Listing 5.4: Forest of directives generated by the preprocessor

```
Directives:
[
  ForDir {
    header: ForHeader {
      var_name: "i",
      range: Range {
        from: "1",
        to: "3",
      },
    },
    body: [
      BaseDir {
        body: "network net{i} cidr(10.0.0.0/24);\n",
      },
      ForDir {
        header: ForHeader {
          var_name: "j",
          range: Range {
            from: "1",
            to: "3",
          },
        },
        body: [
          IfDir {
            header: IfHeader {
              left: "{i} ",
              right: " 1",
              comparison: Equal,
            },
            body: [
              BaseDir {
                body: "host \"server {i} {j}\" interface(net{i});\n",
              },
            ],
            else_body: [
              BaseDir {
                body: "host \"vm {i} {j}\" interface(net{i});\n",
              },
            ],
          },
        ],
      },
    ],
  },
  BaseDir {
    body: "\n",
  },
]
```

Listing 5.5: Output of the preprocessor

```
network net1 cidr(10.0.0.0/24);
    host "server 1 1" interface(net1);
    host "server 1 2" interface(net1);
    host "server 1 3" interface(net1);

network net2 cidr(10.0.0.0/24);
    host "vm 2 1" interface(net2);
    host "vm 2 2" interface(net2);
    host "vm 2 3" interface(net2);

network net3 cidr(10.0.0.0/24);
    host "vm 3 1" interface(net3);
    host "vm 3 2" interface(net3);
    host "vm 3 3" interface(net3);
```

5.1.2 Parser

As described before, the parser takes the output of the preprocessor, parses it, and allocates a the required entities. The main function of the parser in the compilation process, is to verify that the template is syntactically correct and to allocates the structures of the entities that are later used by the validator and for the translation. The entities that can be parsed and created are `Host`, `Network`, `ExtNet`, `Cable` and `Cluster`.

The parsing functions, similarly to the preprocessor ones, are constructed based on `NOM`, which provides many parsing functions that can be combined in order to create the complete parser. As an example of the parsing code, Listing 5.6 shows the high level statement parser function and Listing 5.7 shows the main function that parses Network entities. The parsers for the other entities are constructed in a similar way.

Listing 5.6: Parsing code for statements

```
fn statement<'a, E: ParseError<&'a str> + ContextError<&'a str>>
(i: &'a str) -> IResult<&'a str, Statement, E>
{
    context(
        "statement",
        delimited(
            multispace0,
            alt((
                host,
                network,
                cable,
                ext_net,
                cluster
            )),
            multispace0
        )
    )(i)
}
```

Listing 5.7: Parsing code for Network entity

```
fn network<'a, E: ParseError<&'a str> + ContextError<&'a str>>
(i: &'a str) -> IResult<&'a str, Statement, E>
{
    context(
        "network",
        map(
            tuple((
                tag("network"),
                cut(
                    preceded(
                        multispace1,
                        alt((
                            identifier,
                            quoted_string
                        ))
                    )
                ),
                cut(
                    preceded(
                        multispace1,
                        cidr
                    )
                )
            )),
            |(_, name, cidr)| Statement::Network(
                Network{
                    name: name.to_string(),
                    cidr: cidr.to_string(),
                    ..Default::default()
                }
            )
        )
    )(i)
}
```

```
Parsing error: 0: at line 1, in Digit:
network net cidr(10.0.0.0/260);
                          ^

1: at line 1, in cidr_value:
network net cidr(10.0.0.0/260);
                 ^

2: at line 1, in cidr:
network net cidr(10.0.0.0/260);
            ^

3: at line 1, in network:
network net cidr(10.0.0.0/260);
^

4: at line 1, in statement:
network net cidr(10.0.0.0/260);
^
```

Figure 5.2: Example of error message

One interesting and useful feature provided by `NOM` is the `context` function, which is used often in the code in order to provide relevant error messages generation, similarly to a *calltrace*. Figure 5.2 shows an example of error message that the parser produces when executed with a syntactically incorrect template as input. In the example, the mask of the cidr attribute is not a valid value, because it is not a 8-bit integer.

5.1.3 Validator

The validator is responsible to verify that the topology template is semantically correct. To achieve it, it receives as input the list of entities that the parser allocates and performs multiple checks, including:

- Entity names: it checks that there are no duplicate names, or missing ones. For example, it verifies that when a host is connected to a network, the network

with the given name actually exists and similarly for all the other relationship between entities.

- Correct connection: cables must be connected to two entities and the validator enforces this. Also verifies that, when a cluster is created with an interface on a certain network, all the hosts in the cluster have an interface on the same network.

- IPv4 allocation: it checks that there are no multiple hosts with the same IPv4 address on the same network.

Moreover, the validator also completes the instances with missing information. This can happen for example with the Cable entity, since it possible to specify the two connection both in the cable statement or in the host interface attribute. Therefore, the validator will combine the two information, verify the correctness and add the missing information. Another example is with clusters. Each host in a cluster will receive a virtual IPv4 address on the interfaces used by the cluster. The validator adds this information to the host instances.

After all the validation steps are successful, the validator outputs the list of entities with the updated fields.

5.1.4 Translation

The final step of the compilation from the Description Language to HOT is the translation, which takes the list of entities from the validator and outputs the resulting HOT file.

During this step, the list of entities is iterated, and for each one the method `to_hot()` is called. This function fills a template using the instance data, creating the correct HOT template needed for each entity. This function is implemented slightly differently for every entity type, since some types only have to replace their

name in the template, others have a more complicated logic, such as Host and Cluster.

5.1.5 Extensibility

One of the requirements identified in the design phase was the extensibility to the language. The current implementation of the compiler provides only limited feature regarding this aspect.

One feature currently supported provides the possibility to set environment variables that modify various default values, such as host's and firewall's image and flavor, ExtNet's and cluster's internal network cidr.

Another desired feature is to be able to define derived types, such as the firewall described in Section 4.4.2. The current implementation facilitates this but it still requires editing the compiler code. For example, to implement a new type derived from the host, we only need to add the type to the `HostType` enum, add a function that returns the default values and modify the parsing function to accept also a different type name for the host entity.

Lastly, for creating new types that are not derived from existing ones, some traits and macros are provided to simplify the parser creation for the new entity. With that, it is possible to create a struct that contains as fields all the attributes of the type and use the `derive` attribute with the trait `ParseStatement`. This will provide a default implementation of the `parse` function, which creates a parser that includes all the fields of the struct. However, this requires the type of the field to implement the `ParseAttributeValue` trait, with some common implementations already provided. The result is that to create a parser for the new type it is required only to implement the `ParseAttributeValue` trait for the custom types used in the struct. Therefore, this feature makes it much easier to implement and integrate new types in the compiler.

5.2 Cyber Range infrastructure

As explained before, we use OpenStack as a base infrastructure with some modifications and additional components developed to better adapt it to our needs and integrate the Description Language.

5.2.1 Installation and deployment of OpenStack

Different options and methods are available to install OpenStack. It is possible to install and configure every component manually but it is a very time-consuming method. Therefore, it is suggested to use other deployment systems that automate the installation and configuration of the different components, such as Ansible [34], Packstack [35] and Devstack [36].

The deployment system we use is Packstack, a tool developed by the RDO project, which is a community of people using and deploying OpenStack on CentOS, Fedora, and Red Hat Enterprise Linux [37]. It is particularly useful also during the testing phase because it allows to rapidly deploy OpenStack from a given configuration file that contains multiple options to correctly install and configure the required components. Moreover, it is also possible to extend the OpenStack infrastructure by automatically deploy additional compute nodes when required. Therefore, once the configuration file is created correctly, it is very easy to deploy multiple times or on different machines.

5.2.2 OpenStack modifications

As presented before, we require the use of Layer 2 connections, which OpenStack does not support directly since it does not provide any entity to represent a cable connection. Moreover, it requires all virtual machine's interfaces to have an IP address when connected to a network. In details, Neutron supports allocating networks that

do not have Layer 3 services enabled and the creation of ports without IP addresses associated to them, but Nova, before deploying a virtual machine, verifies that all the ports associated with it have an IP address. To solve this issue, we have to patch Nova to remove this check in certain conditions. This modification was implemented in a pull request [38] that was not merged for lack of interest. Therefore, we rebased the pull request's changes on the current version of the Nova's code and created a patch that we can apply to every compute node that we deploy. The result is that when a cable instance is used in a Description Language template, the compiler can set the property `port_security_enabled` to `false` of virtual machine's ports that are connected to a cable and OpenStack correctly deploys the template. Moreover, when a cable is connected to two virtual machines a dedicated tenant network is created. Instead, when a cable is used to connect a virtual machine to a network, creating an additional network is not required.

The second patch we created slightly modifies the visualization of the topology in the web dashboard. This is needed because we use some entities that are different from those used by OpenStack, or that have a different semantic value. The firewall and cable types of the Description Language are translated to normal virtual machines and networks respectively, with no difference to host and network types. To improve the visualization, the compiler adds tags to the HOT template's entities, which are normally ignored by OpenStack and the web interface. Then, we created a patch for the dashboard that modifies the topology graph, rendering it differently according to the tags associated with the entities.

5.2.3 Cyber Range CLI

Section 5.1 explained how the workflow required to deploy a scenario from a Description Language template requires to first manually compile the template and then upload the resulting HOT template and have OpenStack deploy it.

```
$ cr-cli start simplescenario simple_tests/host_and_net.dl
   [00:00:00] ✔  Compiling the template...
   [00:00:02] ✔  Validating the scenario...
   [00:00:27] ✔  Launching scenario...

Scenario simplescenario:
        Status: CreateComplete
        ID: a0c4c0a8-dd0a-4804-a6fa-5b5ed120f941

        Servers:
                Name: simplescenario-vm-m3gqiwask6nv
                Status: ACTIVE
                ID: c2102ccb-4680-498c-9d8a-60d5106b2c13
                Console: http://10.0.2.15:6080
/vnc_auto.html?path=%3Ftoken%3D09ecef3b-85de-486c-9801-4785b5329d60


Scenario created successfully
```

Figure 5.3: Example of the CLI used to deploy a scenario

In order to simplify this process we created a command line interface (CLI) that automates the various steps. The CLI is also written in `Rust` and interfaces directly with OpenStack using the library `rust-openstack` [39]. However, the library does not provide the functions and structures required to interact with Heat, but only with Nova, Neutron, Glance and Cinder. Therefore, we implemented all the missing components and functions that enable the interactions with the complete Heat APIs.

The result is that the CLI can be used to list, start and stop scenarios on the Cyber Range infrastructure. In particular, to start a scenario it is required to only provide a Description Language template and a name. The CLI will then compile the template, check if the cloud-init configuration files used in the template, if any, are present, verify that the images and flavors used by the virtual machines exist and deploy the topology on OpenStack. Figure 5.3 shows an example execution of the CLI.

Moreover, the CLI can also be used as a `Rust` library and not only as a stand

alone program. This is important for future development because if there will be the need to implement a custom graphical interface then the CLI can be used as library to provide the required interactions with both the Description Language compiler and OpenStack.

6 Testing

This chapter describes the validation testing of our Cyber Range, focusing only on the Description Language, since it is the key component that we developed and that characterize our Cyber Range.

To test the Description Language, we will use the four topologies that we defined in Chapter 4, in order to verify that the Description Language is able to produce a valid result for the requirements of the Cyber Range.

For testing and validating the result, the following parameters will be used:

- **Correctness of translation**. We check that the Description Language can be correctly compiled to a valid HOT template that represent the required topology.

- **Complexity and length** required to write the template using the Description Language. In particular, it is interesting to compare it to the equivalent HOT template, since the new language was designed to be easy to use and to avoid the verbosity of HOT. This test will show whether the implementation of Description Language is effective.

- **Features and complexity comparison** between the Description Language, HOT and other CR templating systems described in Chapter 3. With this test we want to compare what features difference exists between the templating languages and if there are particular topologies or configurations that is not possible to achieve using one templating system or the other.

6.1 Correctness of translation

For the correctness test, for each testing topology, we will analyze what features are required, what is the Description Language template used and check that the result of the compilation can be correctly deployed on OpenStack, generating the desired topology.

Topology 1

The first topology, requires many basic features that are provided by the Description Language . In details, the topology requires the definition of many hosts that are connected to different private networks, each network with different settings. Then, two external networks are used and two hosts that act as Firewalls are defined and connected to both external networks and multiple private networks. Moreover, some hosts requires the use of a specific image and not a generic one. For example, the servers will have a different image than the normal hosts that can use a more generic one.

The template that implements the first topology is shown in Listing 6.1 and includes the definitions for all the elements described above.

Listing 6.1: Description Language template for topology 1

```
// Template for topology 1

ext_net N1;
ext_net N2;

// First set of private networks:
// Management network for smc host
network management cidr(192.168.10.0/24);
host smc image(smc_image) interface(management);

network "intra 1" cidr(192.168.20.0/24);
network "dmz 1" cidr(192.168.30.0/24);

host h1_1 interface("intra 1");
host h1_2 interface("intra 1");

host "server 1"
    image(server_img)
    interface("dmz 1", 192.168.30.11);

firewall FW1
    interface(management, gateway)
    interface("intra 1", gateway)
    interface("dmz 1", gateway)
    interface("N1")
    interface("N2");

network "intra 2"  cidr(192.168.21.0/24);
network "dmz 2"  cidr(192.168.31.0/24);

host h2_1 interface("intra 2");
host h2_2 interface("intra 2");

host "server 2"
    image(server_img)
    interface("dmz 2", 192.168.31.12);

firewall FW2
    interface("intra 2", gateway)
    interface("dmz 2", gateway)
    interface("N1")
    interface("N2");
```

The template can be correctly compiled to HOT and deployed on OpenStack. The final deployed topology can be seen in Figure 6.1.

Topology 2

The second topology, is very similar to the first one but in addition it requires the cluster feature. It includes two clusters, each composed by two firewalls. The other elements, networks and hosts, are identical.

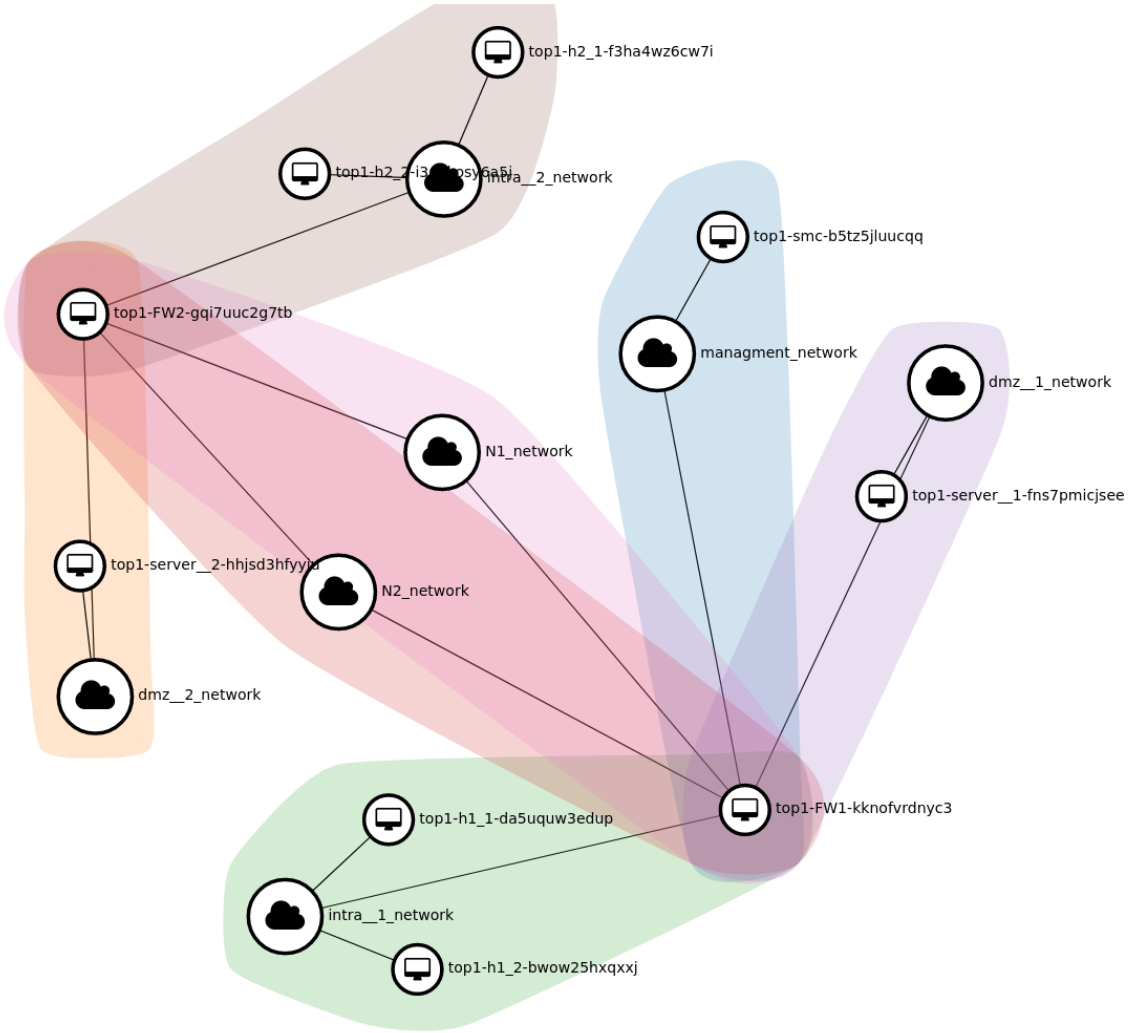

Figure 6.1: Topology 1 deployed on OpenStack

The second topology is implemented as shown in Listing 6.2. Each firewall has two IPv4 addresses assigned to each interface, one specified by the interface attribute in the firewall entity and the second one, the gateway IPv4 address, is assigned by the cluster, since the cluster instance assigns the IPv4 address specified in the interface attribute to each host composing the cluster.

Listing 6.2: Description Language template for topology 2

```
ext_net N1;
ext_net N2;

network management cidr(192.168.10.0/24);
host smc image(smc_image) interface(management);

network "intra 1" cidr(192.168.20.0/24);
network "dmz 1" cidr(192.168.30.0/24);

host h1_1 interface("intra 1");
host h1_2 interface("intra 1");

host "server 1" image(server_img)  interface("dmz 1", 192.168.30.21);

firewall FW1_1
    interface(management, 192.168.10.11)
    interface("intra 1", 192.168.20.11)
    interface("dmz 1", 192.168.30.11)
    interface("N1")
    interface("N2");

firewall FW1_2
    interface(management, 192.168.10.12)
    interface("intra 1", 192.168.20.12)
    interface("dmz 1", 192.168.30.12)
    interface("N1")
    interface("N2");

cluster cluster1
    firewall(FW1_1)
    firewall(FW1_2)
    interface(management, gateway)
    interface("intra 1", gateway)
    interface("dmz 1", gateway);


network "intra 2"  cidr(192.168.21.0/24);
network "dmz 2"  cidr(192.168.31.0/24);

host h2_1 interface("intra 2");
host h2_2 interface("intra 2");

host "server 2" image(server_img) interface("dmz 2", 192.168.31.22);

firewall FW2_1
    interface("intra 2", 192.168.21.11)
    interface("dmz 2", 192.168.31.11)
    interface("N1")
    interface("N2");

firewall FW2_2
    interface("intra 2", 192.168.21.12)
    interface("dmz 2", 192.168.31.12)
    interface("N1")
    interface("N2");

cluster cluster2
    firewall(FW2_1)
    firewall(FW2_2)
    interface("intra 2", gateway)
    interface("dmz 2", gateway);
```

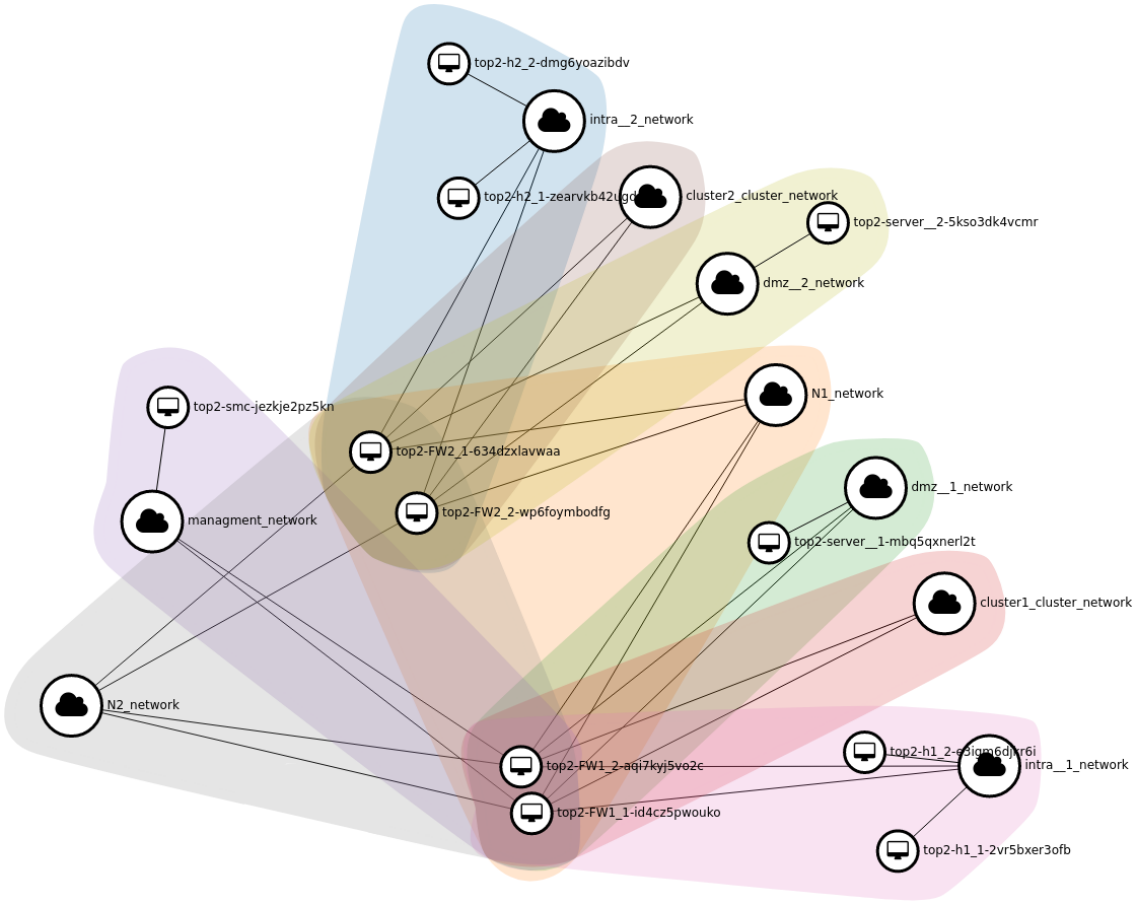

Figure 6.2: Topology 2 deployed on OpenStack

The template can be correctly compiled to HOT and deployed on OpenStack . The final topology deployed on OpenStack can be seen in the diagram in Figure 6.2. For the two cluster entities, an additional network is created, *cluster1_ cluster_ network* and *cluster2_ cluster_ network* in the Figure, with all the hosts and firewalls that compose the cluster connected to it with an additional interface. In the second topology only firewalls are part of the clusters but that is not a requirement.

Topology 3

The third topology is quite different from the previous ones and it is used to test the performance of the security components in a high traffic situation. Therefore,

the topology of the private networks is simpler, but more networks and hosts are included. Therefore, no additional features are required to describe this topology, other than those already presented in the previous tests.

The third topology is described in the template in Listing 6.3.

Listing 6.3: Description Language template for topology 3

```
ext_net N1;

network management cidr(192.168.100.0/24);
host smc image(smc_image) interface(management);

network "intra 1" cidr(192.168.10.0/24);
host h1 interface("intra 1");
firewall FW1
    interface("management", gateway)
    interface("intra 1", gateway)
    interface("N1");


network "intra 2" cidr(192.168.20.0/24);
host h2 interface("intra 2");
firewall FW2
    interface("intra 2", gateway)
    interface("N1");


network "intra 3" cidr(192.168.30.0/24);
host h3 interface("intra 3");
firewall FW3
    interface("intra 3", gateway)
    interface("N1");


network "intra 4" cidr(192.168.40.0/24);
host h4 interface("intra 4");
firewall FW4
    interface("intra 4", gateway)
    interface("N1");


network "intra 5" cidr(192.168.50.0/24);
host h5 interface("intra 5");
firewall FW5
    interface("intra 5", gateway)
    interface("N1");
```

The template can be correctly compiled to HOT and deployed on OpenStack. The final topology deployed on OpenStack can be seen in the diagram in Figure 6.3.

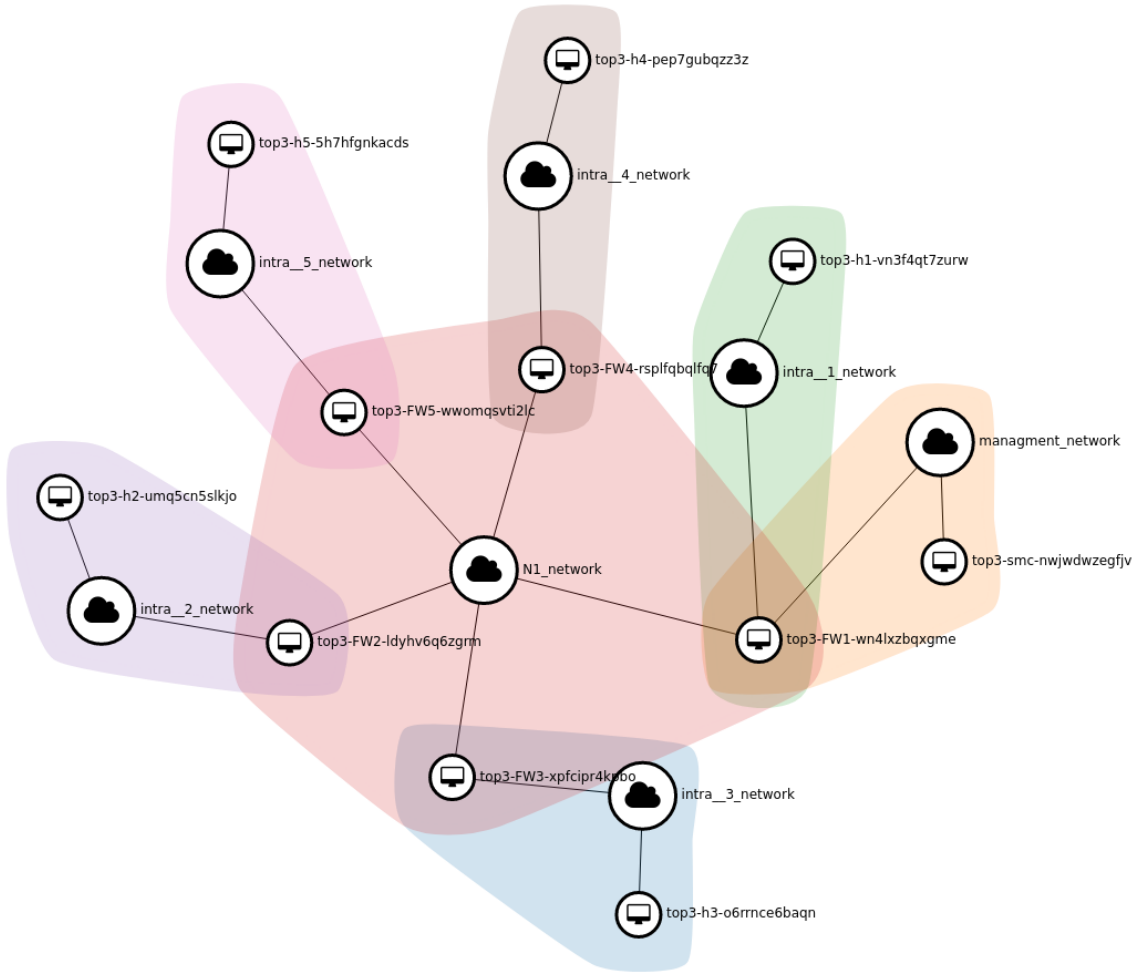

Figure 6.3: Topology 3 deployed on OpenStack

Topology 4

The last topology includes two Layer 2 IPS/IDS which are placed between the Firewalls and the external network. In order to use a layer 2 host, the topology requires the use of layer 2 connections. The Description Language provides the layer 2 connection features with the cable entity, which is used four times in the template shown in Listing 6.4, to connect both IPS/IDS to the Firewall and the external networks.

Listing 6.4: Description Language template for topology 4

```
ext_net N1;

network management cidr(192.168.10.0/24);
host "smc"  image(smc_image)  interface(management);

network "intra 1"  cidr(192.168.20.0/24);
network "dmz 1"  cidr(192.168.30.0/24);

host "h1" interface("intra 1");
host "server 1"
    image(server_img)
    interface("dmz 1", 192.168.30.11);

cable ips1_ext connection(N1);
cable ips_fw_1;

host "ips1"
    image(ips_img)
    interface(ips1_ext)
    interface (ips_fw_1);

firewall FW1
    interface(ips_fw_1)
    interface("management", gateway)
    interface("intra 1", gateway)
    interface("dmz 1", gateway);


network "intra 2"  cidr(192.168.20.0/24);
network "dmz 2"  cidr(192.168.30.0/24);

host "h2" interface("intra 2");
host "server 2"
    image(server_img)
    interface("dmz 2", 192.168.30.12);

cable ips2_ext connection(N1);
cable ips_fw_2;

host "ips2"
    image(ips_img)
    interface(ips2_ext)
    interface (ips_fw_2);

firewall FW2
    interface(ips_fw_2)
    interface("intra 2", gateway)
    interface("dmz 2", gateway);
```

The template can be correctly compiled to HOT and deployed on OpenStack. The final topology deployed on OpenStack can be seen in the diagram in Figure 6.4. The cable entities are translated to normal networks but without any IPv4 allocation pools, since there is no Layer 2 connection feature in OpenStack, when the cable

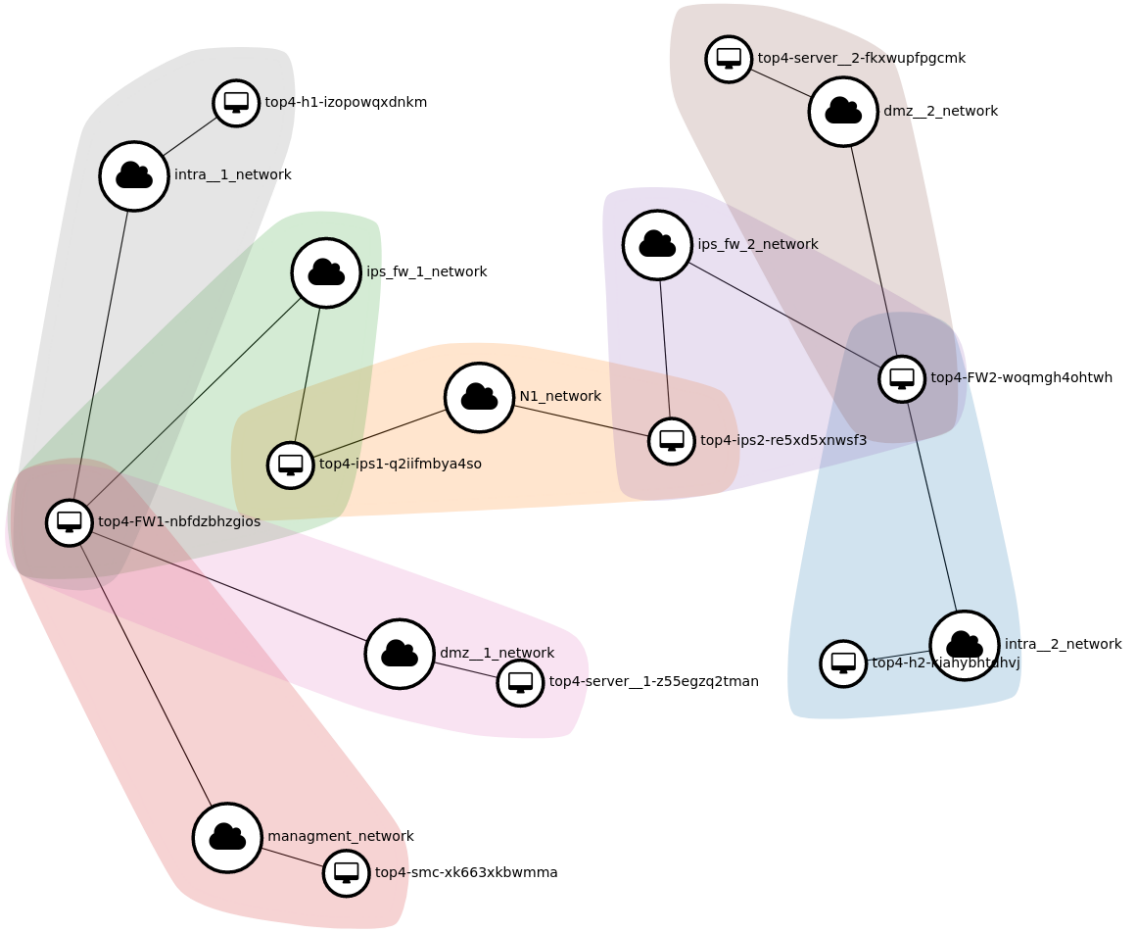

Figure 6.4: Topology 4 deployed on OpenStack

connects two hosts, like the connection between the IPS/IDS and the Firewall. Instead, when the cable connects a host to a network, a port without an IPv4 address is created on the given network and associated to the host. This second variant of the implementation is used to connect the IPS/IDS to the external network.

6.2 Complexity and length

To evaluate the complexity of the Description Language, for each testing topology we analyze the features used in the template, the number of entities and lines, and compare it to the equivalent HOT template. Moreover, we compare both a template

using only the Description Language entities and a template that makes also use of the preprocessor, which should reduce the complexity and length.

For the first topology, Listing 6.1 uses the host, network and extnet entity types, with a total of 16 statements used. Listing 6.5 shows an equivalent template that also uses the preprocessor features, which reduce the number of entities from 16 to 9 but also introduces 4 directives. The compilation result uses network, subnet, port and host resources types, with a total of 30 resources declared. If we consider the number of lines used in the three templates, they have 42, 31 and 244 lines respectively.

Listing 6.5: Description Language template for topology 1 with the use of preprocessor directives.

```
1  ext_net N1;
2  ext_net N2;
3
4  network management  cidr(192.168.10.0/24);
5  host smc  image(smc_image)  interface(management);
6
7  #for i (1,2):
8      network "intra {i}"  cidr(192.168.20.0/24);
9      network "dmz {i}"  cidr(192.168.30.0/24);
10
11     #for j (1,2):
12     host "h{i}_{j}"
13         #if ({j} = 2):
14         image(custom_img)
15         #endif
16         interface("intra {i}");
17     #endfor
18
19     host "server {i}"
20         image(server_img)
21         interface("dmz {i}", 192.168.30.1{i});
22
23     firewall FW{i}
24         #if ({i} = 1):
25         interface("management", gateway)
26         #endif
27         interface("intra {i}", gateway)
28         interface("dmz {i}", gateway)
29         interface("N1")
30         interface("N2");
31 #endfor
```

For the second topology, Listing 6.2 uses the same entity types of the first topology plus the cable, with a total of 20 statements used. Listing 6.6 shows the result

of the previous template with the addition of preprocessor directives, which uses 10 entities and 5 directives. The equivalent HOT template uses the same type of resources as the previous topology since there is no cable resource type available in HOT, with a total of 41 resources declared. The number of lines used are 61, 38 and 351 lines respectively.

Listing 6.6: Description Language template for topology 2 with the use of preprocessor directives.

```
ext_net N1;
ext_net N2;

network management  cidr(192.168.10.0/24);
host smc image(smc_image) interface(management);

#for i (1,2):
    network "intra {i}"  cidr(192.168.20.0/24);
    network "dmz {i}"  cidr(192.168.30.0/24);

    #for j (1,2):
    host "h{i}_{j}" interface("intra {i}");
    #endfor

    host "server {i}"
        image(server_img)
        interface("dmz {i}", 192.168.30.2{i});

    #for j (1,2):
        firewall FW{i}_{j}
            #if ({i} = 1):
            interface("management", 192.168.10.1{j})
            #endif
            interface("intra {i}", 192.168.20.1{j})
            interface("dmz {i}", 192.168.30.1{j})
            interface("N1")
            interface("N2");
    #endfor

    cluster cluster{i}
        firewall(FW{i}_1)
        firewall(FW{i}_2)
        #if ({i} = 1):
        interface("management", gateway)
        #endif
        interface("intra {i}", gateway)
        interface("dmz {i}", gateway);
#endfor
```

Topology 3 does not introduce new entity types but only increases the number of them to 18. With this topology, the impact of the preprocessor directives to reduce the length of the template is more visible, as shown in Listing 6.7 which contains

only 6 entities and 2 directives. Instead, the HOT template uses 31 resources.

Listing 6.7: Description Language template for topology 3 with the use of preprocessor directives.

```
ext_net N1;

network management cidr(192.168.100.0/24);
host smc  image(smc_image)  interface(management);

#for i (1,5):
    network "intra {i}" cidr(192.168.{i}0.0/24);
    host "h{i}" interface("intra {i}");

    firewall FW{i}
        #if ({i} = 1):
        interface("management", gateway)
        #endif
        interface("intra {i}", gateway)
        interface("N1");
#endfor
```

For the last topology, the original Description Language template in Listing 6.4 uses the new entity type cluster and has a total of 19 statements. The version with the preprocessor directives, reduces the number of entities to 11, plus 2 directives, as shown in Listing 6.8. The HOT template, does not use additional resource types and contains 36 resources.

Listing 6.8: Description Language template for topology 4 with the use of preprocessor directives.

```
ext_net N1;

network management cidr(192.168.10.0/24);

host smc  image(smc_image)  interface(management);

#for i (1,2):
    network "intra {i}"  cidr(192.168.20.0/24);
    network "dmz {i}"  cidr(192.168.30.0/24);

    host "h{i}" interface("intra {i}");
    host "server {i}" image(server_img)
        interface("dmz {i}", 192.168.30.1{i});

    cable ips{i}_ext connection(N1);

    cable ips_fw_{i};

    host "ips{i}"
        image(ips_img)
        interface(ips{i}_ext)
        interface (ips_fw_{i});

    firewall FW{i}
        interface(ips_fw_{i})
        #if ({i} = 1):
        interface("management", gateway)
        #endif
        interface("intra {i}", gateway)
        interface("dmz {i}", gateway);
#endfor
```

Table 6.1 provides a summary of the comparison between the three templates for all four topologies. It is clear that the Description Language templates, both with and without the use of the preprocessor, require much less lines of code and uses also less entities, especially in the case of the templates that use also the preprocessor.

6.3 Features supported

The last evaluation that we propose for the Description Language templating system is a feature comparison with HOT and also the existing systems presented in Chapter 3.

In the previous section we shown that the Description Language can describe

Topology	Description Language		DL + preprocessor		HOT	
	lines	entities	lines	entities	lines	entities
1	42	16	31	9 + 4 dir	244	30
2	61	20	38	10 + 5 dir	351	41
3	39	18	16	6 + 2 dir	251	31
4	48	19	31	11 + 2 dir	268	36

Table 6.1: COMPLEXITY SUMMARY

topologies with a much less verbose template than HOT. This is possible because many details are not required, for example it is not needed to create both *network* and *subnet* entities. This reduces the flexibility of the Description Language and it is not possible to configure and control certain aspects of the topology. However, HOT was designed for a different use-case, the deployment of cloud infrastructure within OpenStack that might require a set of configuration options that are not needed when using a Cyber Range.

Moreover, the Description Language provides additional features that are not present in HOT. The *cable* and *cluster* entities are not available with HOT and their implementation require the configuration of multiple entities that make the resulting template more complex and verbose. In addition, the preprocessor provides a more elastic mechanism to extend the template than the one provided by HOT, which can be used only within a single entity and is more verbose.

Overall, with the Description Language it is possible to write more compact templates, with only the set of features that we defined as relevant for our Cyber Range during the design phase, described in Chapter 4. On the other hand, HOT is a more verbose language that can control and configure certain aspect of the topology more in detail, but with limited effective usefulness in the context of Cyber Ranges. In fact, in Chapter 3 we presented various existing Cyber Ranges that use OpenStack as a base infrastructure and most of those decided to implement and use a different

templating system than HOT.

We continue the evaluation of the Description Language with the comparison with the templating solutions of the Cyber Ranges presented in Chapter 3. Table 6.2 shows a summarized comparison of the different systems, similar to Table 3.1 but with the addition of HOT and our solution. As mentioned in Section 3.2, we decided to compare the Description Language only with the Cyber Ranges that provide a clear description of their templating system and that provide a more generic use-case, which is not bound to very specific systems like the *Autonomous vehicle shuttles* [30] Cyber Range.

CRACK templating system uses YAML syntax, resulting in an enchanted version of HOT. Therefore, it supports all the features of HOT plus additional entities and properties to configure software configurations and other aspects. This, combined with the formal verification, allows to better and more explicitly describe the configuration of the virtual machines than our solution, which is based on `cloud-init` scripts. Being similar to HOT, the language derives also the same complexity and verbosity issues. Lastly, it supports element types derivation more generally than our solution that has only limited support.

Kypo uses YAML as well, but with a different structure and less details, which makes templates more compact than those produced with HOT but still more verbose than our Description Language. The topologies that can be described are more limited than HOT and our solution, due to the limited elements available for the topology. For the software configuration of hosts, it uses separate Ansible files combined with a system to deploy them over SSH, which is an approach comparable to `cloud-init` used in our solution.

Similarly to the previous solutions, **Norway 1** uses a YAML based file but with a different structure. It supports only two elements to define the topology, Nodes, representing virtual machines, and Routers, which define the sub-networks to which

they are connected. Moreover, all the Routers are connected to a common network, causing a much more limited structure in the topology that can be defined than what is possible to do with our Description Language. However, **Norway 1** provides a more explicit software configuration of the virtual machines by the use of Service, Vulnerability and Challenge elements, that are then used to create Ansible files. One feature in common with our solution is the use of both syntax and semantic validation.

Norway 2 is the only solution that uses JSON templates with two elements that can be used: Machine and Subnet. This is probably the system that has the biggest limitations since a Machine can be connected to a single Subnet. Moreover, it requires some manual interactions with OpenStack as it uses the ID of a network that has to be already deployed. The authors also provide an example where a large topology is deployed, which required to use multiple times the same template and then some manual work to interconnect all the topologies created. This shows how our preprocessor component can be very useful to automate the creation of large topologies without having to deploying multiple times the same template. One aspect where this solution provides a better value than others, is the possibility to model attackers and defenders actions, including also a traffic generator, which is something that was not yet developed in our solution.

VSDL uses a custom grammar for the templates and it is the only solution that allows to use logical connectives to define the topology elements instead of hard-coded values. Another unique feature is the use of time-base modifiers that can modify the proprieties of the deployed topology during the execution of the exercise. Also in this case, there is no support for easing the creation of topologies with a high number of nodes and networks.

The **Description Language** we propose, also uses a custom grammar, which was specifically designed to reduce the verbosity and complexity of the topology

templates. It provides three base elements: Host, Network and Cable, and three additional elements: Ext_Net, Cluster and Firewall. The additional elements help by providing shorter definitions of certain specific topology elements or configurations, as well as increased semantic clarity. With the Description Language it should be possible to model various kinds of topology, without limitations. Moreover, the Cable element is a feature unique to our language and it provides the ability to describe topologies that are not possible to model with the other solutions, but it also required a small modification of the OpenStack infrastructure. The validations of the scenario that we provide are based on the syntax and semantic, similarly to the Norway 1 solution. For the software configuration of the hosts, the language supports only the use of scripts that will be executed by `cloud-init`, without the possibility of modelling the software or vulnerabilities deployed directly in the language and thus not performing semantic checks on this aspect. Another unique feature of our solution is the use of a preprocessor to simplify the definition of similar entities, which is especially useful in the case of large network with repeated structures in the topology, as described before.

Cyber Range	Language format	Topology elements	Verification	Host configuration
CRACK	YAML	Similar to HOT: Host, Network, Subnet, Port	Formal verification with Datalog	Language elements to model Software, Vulnerabilities, Users and more.
Kypo	YAML	Host, Router, Network and network, router mappings	Not specified	Ansible YAML configuration files
Norway 1	YAML	Node and Router. Subnets are defined by routers	Syntax validation and semantic validation	Language elements to model Software, Vulnerability, Challenge, Teams and more
Norway 2	JSON	Subnet, Machine	Formal verification with Datalog	Language elements to model Services, Vulnerabilities, Challenge
VSDL	Custom grammar	Network and Node	Formal verification with CVC4	Software and vulnerabilities properties
HOT	YAML	Network, Subnet, Port, Server, and more	Syntax and semantic validation when deployed	cloud-init scripts
Description Language	Custom grammar with preprocessor	Host, Network, Cable and additional elements: Firewall, External Network and Cluster	Syntax and semantic validation	cloud-init scripts

Table 6.2: COMPARISON SUMMARY

7 Conclusion and Future Work

7.1 Conclusion

Cyber Ranges are becoming a fundamental component for security trainings, exercises and testing, with an increasing interest also due to the increasing request for cyber security experts, that can be seen also in the growing number of research on the topic. When a Cyber Range is used, the most time-consuming activity is the development of the scenario, which requires to define the virtual topology that will be deployed and the configurations of the various virtual machines that will be used.

Our work focuses on this aspect and aims to ease and simplify the process of defining the scenario and in particular the virtual topology, by developing a Cyber Ranges that makes use of a new templating system.

First we analyzed a selection of the currently available Cyber Ranges and in particular the templating systems and languages available for the definition of scenarios.

Then, we started by defining the requirements of the Cyber Range and the Description Language, which were used to define four base topologies that the system should support. The four topologies were used to describe the practical features required. For the Cyber Range infrastructure we identified OpenStack as a good base platform, that can be extended and improved with a new Description Language. From the requirements identified, we created the grammar of the new language, that

was presented with some examples.

The implementation chapter focused on the Description Language compiler and its components, that were created using `Rust` and the `NOM` library. Moreover, we presented also the other components that we created, a command line interface, which can be used to deploy topologies on OpenStack from the template file, and the patches for OpenStack that we require to enable certain features of the Description Language, in particular the Cable entity.

The implementation, and in particular the language, was tested to demonstrate that it can correctly describe the four topologies identified during the requirements definition and that it is possible to correctly deploy them on OpenStack. Moreover, we showed that the length and complexity of the template are reduced when using our solution instead of HOT. Lastly, we compared our proposed solution to the existing languages that were presented before, showing that the Description Language supports unique features that are not present in the other approaches, such as the Cable entity for Layer 2 connections and the preprocessor that can be used to simplify the creation of large topologies.

However, the Description Language does not provides a function to configure the virtual machines deployed in the topology directly, but it makes use of start-up scripts that are executed by cloud-init.

To conclude, the tests and the comparison presented in Chapter 6 show that our solution can be used to correctly deploy topologies and the Description Language is able to lower the complexity and length of the required template, which can reduce the time required for the operators of the Cyber Range to setup scenarios. Moreover, our work was also presented at the fourth *workshop on Cyber Range Technologies and Applications* (CACOE'22) organized in conjunction with *IEEE EuroS&P 2022* [40].

However, there are aspects that require improvements and additional work, which are presented in the next section.

7.2 Future work

There are different aspects of the Description Language that we plan to improve in the future.

First, the extensibility of the language is still limited. It is possible to create custom types based on existing ones, but with different default values, like the firewall type, but this still requires to change the code of the compiler. An improvement would be to enable the creation of derived types in a configuration file, so that it is not required to change the code. Moreover, the definition of new types is also important. In Chapter 5 we described the use of Rust traits to automatically generate the parsing functions. This can be very useful when new types are required, since it requires to only implement the validation and translation steps. The last feature to increase the extensibility that we are planning, is the possibility to import another file in a template. This would allow to divide the template in multiple files and to provide common topology components that can be used without the need of redefining it for each template.

One aspect that the comparison in Chapter 6 outlined, is the limited support for configuring virtual machines in the Description Language. Currently, only cloud-init scripts are supported, which can be limiting because cloud-init is not available for Windows systems and also it is not possible to define and validate the configuration options using elements of the language, which is a feature of some of the existing Cyber Ranges, presented in Chapter 3. A first step that can improve this aspect, would be the creation of a repository or database of initialization scripts, which would allow users to use predefined configurations without the need to write cloud-init scripts themselves for common operations. Alternatively, a different configuration system could be used, but this option will require more research and modification of the current implementation.

Another topic that we want to work on, is the automation of the collaboration

process between different organizations, interconnecting the Cyber Ranges they use. This feature is not present in the analyzed Cyber Ranges and would be useful for multiple reasons. First, it enables to extend the virtualization capabilities when a scenario requires too many resources, without the need of extending the Cyber Range hardware. Secondly, for projects that involve multiple organizations it would be very convenient to connect different Cyber Ranges instead of having to develop the entire scenario using a single system. Moreover, when proprietary virtual machines are required in a scenario developed by multiple organizations, it would be possible to provide the virtual machine to others by deploying on the organization own Cyber Range, without the need to provide the machine image or source code. One possible solution is to define virtual networks that span on multiple Cyber Ranges using VPN connections to make this extension transparent to the virtual environment. This feature will require some modifications to the Description Language and also the implementation of additional components for the virtualization infrastructure because OpenStack does not provide this capability explicitly. OpenStack supports the creation of IPsec connections with the VaaS component but it is not sufficient since there would be the need also for a coordination mechanism that could automatically create and manage connections with different Cyber Ranges.

To conclude, we are studying and planning various improvements for the Description Language and the implementation is planned to be part of the modules used and tested in a future pilot scenario of the Cyber-MAR H2020 [41] project, that will validate the functionality and usefulness in a real word exercise.

References

[1] C. Nast, "They Told Their Therapists Everything. Hackers Leaked It All", *WIRED*, May 2021. [Online]. Available: https://www.wired.com/story/vastaamo-psychotherapy-patients-hack-data-breach.

[2] S. Kumar, S. Gupta, and S. Arora, "Research trends in network-based intrusion detection systems: A review", *IEEE Access*, vol. 9, pp. 157 761–157 779, 2021. DOI: 10.1109/ACCESS.2021.3129775.

[3] R. Nakata and A. Otsuka, "CyExec*: A high-performance container-based cyber range with scenario randomization", *IEEE Access*, vol. 9, pp. 109 095–109 114, 2021. DOI: 10.1109/access.2021.3101245.

[4] M. M. Yamin, B. Katt, and V. Gkioulos, "Cyber ranges and security testbeds: Scenarios, functions, tools and architecture", *Computers & Security*, vol. 88, p. 101 636, Jan. 2020. DOI: 10.1016/j.cose.2019.101636.

[5] E. C. Chaskos, *Cyber-security training: A comparative analysis of cyber-ranges and emerging trends*, Mar. 2019.

[6] K. B. Vekaria, P. Calyam, S. Wang, R. Payyavula, M. Rockey, and N. Ahmed, "Cyber range for research-inspired learning of "attack defense by pretense" principle and practice", *IEEE Transactions on Learning Technologies*, vol. 14, no. 3, pp. 322–337, Jun. 2021. DOI: 10.1109/tlt.2021.3091904.

[7] J. Vykopal, M. Vizvary, R. Oslejsek, P. Celeda, and D. Tovarnak, "Lessons learned from complex hands-on defence exercises in a cyber range", in *2017 IEEE Frontiers in Education Conference (FIE)*, IEEE, Oct. 2017. DOI: 10.1109/fie.2017.8190713.

[8] M. M. Yamin, B. Katt, and M. Nowostawski, "Serious games as a tool to model attack and defense scenarios for cyber-security exercises", *Computers & Security*, vol. 110, p. 102 450, Nov. 2021. DOI: 10.1016/j.cose.2021.102450.

[9] T. Gustafsson and J. Almroth, "Cyber range automation overview with a case study of CRATE", in *Secure IT Systems*, Springer International Publishing, 2021, pp. 192–209. DOI: 10.1007/978-3-030-70852-8_12.

[10] *Open Source Cloud Computing Infrastructure - OpenStack*, [Online; accessed 29. Apr. 2022], Apr. 2022. [Online]. Available: https://www.openstack.org.

[11] *Chapter 1. Components Red Hat OpenStack Platform 9 | Red Hat Customer Portal*, [Online; accessed 28. Apr. 2022], Apr. 2022. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/9/html/architecture_guide/components.

[12] *Messaging that just works — RabbitMQ*, [Online; accessed 29. Apr. 2022], Apr. 2022. [Online]. Available: https://www.rabbitmq.com.

[13] *OpenStack Docs: Yoga Services and Libraries*, [Online; accessed 2. May 2022], Apr. 2022. [Online]. Available: https://docs.openstack.org/yoga/projects.html.

[14] *Open vSwitch*, [Online; accessed 4. May 2022], Apr. 2022. [Online]. Available: https://www.openvswitch.org.

[15] F. Callegati, W. Cerroni, and C. Contoli, "Virtual networking performance in OpenStack platform for network function virtualization", *Journal of Electrical and Computer Engineering*, vol. 2016, pp. 1–15, 2016, ISSN: 2090-0147. DOI:

10.1155/2016/5249421. [Online]. Available: https://doi.org/10.1155/2016/5249421.

[16] T. W. Edgar and T. R. Rice, "Experiment as a service", in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, IEEE, Apr. 2017, pp. 1–6. DOI: 10.1109/ths.2017.7943470.

[17] *Haystack, version 00*, Apr. 2017. [Online]. Available: https://www.osti.gov/biblio/1349748.

[18] E. Russo, G. Costa, and A. Armando, "Building next generation cyber ranges with CRACK", *Computers & Security*, vol. 95, p. 101 837, Aug. 2020. DOI: 10.1016/j.cose.2020.101837.

[19] *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC | OASIS*, [Online; accessed 10. Mar. 2022], Mar. 2022. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[20] G. Gottlob, S. Ceri, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)", *IEEE Transactions on Knowledge & Data Engineering*, vol. 1, no. 01, pp. 146–166, Jan. 1989, ISSN: 1558-2191. DOI: 10.1109/69.43410.

[21] J. Vykopal, P. Celeda, P. Seda, V. Svabensky, and D. Tovarnak, "Scalable learning environments for teaching cybersecurity hands-on", in *2021 IEEE Frontiers in Education Conference (FIE)*, IEEE, Oct. 2021. DOI: 10.1109/fie49875.2021.9637180.

[22] M. M. Yamin and B. Katt, "Modeling and executing cyber security exercise scenarios in cyber ranges", *Computers & Security*, vol. 116, p. 102 635, May 2022. DOI: 10.1016/j.cose.2022.102635.

[23] G. Costa, E. Russo, and A. Armando, "Automating the generation of cyber range virtual scenarios with VSDL", *CoRR*, vol. abs/2001.06681, 2020. arXiv: 2001.06681. [Online]. Available: https://arxiv.org/abs/2001.06681.

[24] C. Barrett, C. L. Conway, M. Deters, *et al.*, "Cvc4", in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177, ISBN: 978-3-642-22110-1.

[25] J. Almroth and T. Gustafsson, "CRATE exercise control – a cyber defense exercise management and support tool", in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, Sep. 2020. DOI: 10.1109/eurospw51379.2020.00014.

[26] R. Beuran, D. Tang, C. Pham, K.-i. Chinen, Y. Tan, and Y. Shinoda, "Integrated framework for hands-on cybersecurity training: CyTrONE", *Computers & Security*, vol. 78, pp. 43–59, Sep. 2018. DOI: 10.1016/j.cose.2018.06.001.

[27] R. Beuran, T. Inoue, Y. Tan, and Y. Shinoda, *Realistic cybersecurity training via scenario progression management*, Jun. 2019. DOI: 10.1109/eurospw.2019.00014.

[28] R. Chadha, T. Bowen, C.-Y. J. Chiang, *et al.*, "CyberVAN: A cyber security virtual assured network testbed", in *MILCOM 2016 - 2016 IEEE Military Communications Conference*, IEEE, Nov. 2016. DOI: 10.1109/milcom.2016.7795481.

[29] N. Herold, M. Wachs, M. Dorfhuber, C. Rudolf, S. Liebald, and G. Carle, *Achieving Reproducible Network Environments with INSALATA*, D. Tuncer, R. Koch, R. Badonnel, and B. Stiller, Eds. Springer International Publishing, 2017, pp. 30–44. DOI: 10.1007/978-3-319-60774-0_3.

[30] A. Roberts, O. Maennel, and N. Snetkov, "Cybersecurity test range for autonomous vehicle shuttles", in *2021 IEEE European Symposium on Security*

and Privacy Workshops (EuroS&PW), IEEE, Sep. 2021. DOI: 10.1109/eurospw54576.2021.00031.

[31] *Heat Orchestration Template (HOT) specification — openstack-heat 18.1.0.dev3 documentation*, [Online; accessed 18. May 2022], Apr. 2022. [Online]. Available: https://docs.openstack.org/heat/latest/template_guide/hot_spec.html.

[32] Geal, *nom*, [Online; accessed 31. May 2022], May 2022. [Online]. Available: https://github.com/Geal/nom.

[33] rust-lang-nursery, *lazy-static.rs*, [Online; accessed 31. May 2022], May 2022. [Online]. Available: https://github.com/rust-lang-nursery/lazy-static.rs.

[34] R. H. Ansible, *Ansible is Simple IT Automation*, [Online; accessed 14. Mar. 2023], Mar. 2023. [Online]. Available: https://www.ansible.com.

[35] *Packstack — RDO*, [Online; accessed 14. Mar. 2023], Jan. 2023. [Online]. Available: https://www.rdoproject.org/install/packstack.

[36] *DevStack — DevStack documentation*, [Online; accessed 14. Mar. 2023], Mar. 2023. [Online]. Available: https://docs.openstack.org/devstack/latest.

[37] *RDO*, [Online; accessed 3. Jun. 2022], Jan. 2022. [Online]. Available: https://www.rdoproject.org.

[38] *Allow vms to use unaddressed ports (ic622f9a9)*, [Online; accessed 10. Mar. 2022], Mar. 2022. [Online]. Available: https://review.opendev.org/c/openstack/nova/+/533249.

[39] dtantsur, *rust-openstack*, [Online; accessed 3. Jun. 2022], Jun. 2022. [Online]. Available: https://github.com/dtantsur/rust-openstack.

[40] A. Dalla Costa and J. Kuusijärvi, "Programmatic description language for cyber range topology creation", in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022, pp. 403–412. DOI: 10.1109/EuroSPW55150.2022.00048.

[41] O. Jacq, P. G. Salazar, K. Parasuraman, *et al.*, "The Cyber-MAR project: First results and perspectives on the use of hybrid cyber ranges for port cyber risk assessment", in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2021, pp. 409–414. DOI: 10.1109/CSR51186.2021.9527968.