UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN
INGENIERÍA DE REDES Y SERVICIOS TELEMÁTICOS

TRABAJO FIN DE MÁSTER

DESIGN OF AN ARCHITECTURE FOR CYBER-ATTACK DETECTION ON AN SDN ENVIRONMENT

FERNANDO BENAYAS DE LOS SANTOS

2019

TRABAJO DE FIN DE MÁSTER

Título:	Diseño de una arquitectura para detección de ciberataques en un entorno SDN.
Título (inglés):	Design of an architecture for cyber-attack detection on an SDN environment.
Autor:	Fernando Benayas de los Santos
Tutor:	Álvaro Carrera Barroso
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO DE FIN DE MÁSTER

Design of an architecture for cyber-attack detection on an SDN environment.

Julio 2019

Resumen

Esta memoria es el resultado de un proyecto cuyo objetivo ha consistido en el diseño de una arquitectura para un sistema de detección de ataques Denial of Service sobre redes definidas por software (SDN) y su implementación haciendo uso de herramientas de tratamiento Big Data, tecnologías de web semántica y técnicas de clasificación de Machine Learning en el marco del trabajo realizado por el Grupo de Sistemas Inteligentes. Para llevar a cabo este proyecto, se ha desarrollado un módulo de almacenamiento de datos a partir del cual se desarrolla un módulo de recolección e inyección de datos en dicho sistema de almacenamiento, un sistema de análisis de datos almacenados en busca de ataques y un sistema de semantización de dichos datos almacenados.

Se ha elegido un caso de uso específico que se ha implementado mediante el uso de tecnologías de virtualización de redes SDN, de tal manera que no fuera necesario un despliegue completo de red para comprobar el funcionamiento del sistema antes descrito. Dicha red es gestionada por un controlador OpenDaylight, a través del cual realizaremos cualquier interacción relacionada con la gestión de ataques. Para la detección de estos ataques utilizaremos modelos de Machine Learning previamente aprendidos mediante datasets de ciberataques anotados. Estos modelos serán capaces de detectar patrones de ataque en datos acerca del tráfico de red, e informarán de dichos ataques al tiempo que toman acciones para bloquearlos. Los datos tanto recolectados como generados serán semantizados según una ontología desarrollada en este proyecto, con el objetivo de que los datos sean fácilmente accesibles.

Por último, se desarrollará un sistema de visualización que permitirá consultar información tanto sobre la red, como sobre los posibles ataques que se detecten, así como variables acerca de dichos ataques. También permitirá la ejecución directa de queries semánticas para extraer datos específicos. Como resultado, obtendremos un sistema adaptador para redes SDN que permite la detección de ataques Denial of Service, la recuperación autónoma ante dichos ataques sin afectar al servicio o a otros clientes y la estructuración de los datos mediante una ontología y la visualización del proceso.

Palabras clave: SDN, Big Data Analytics, Linked Data, SPARQL, Elastic, Opendaylight, Machine Learning, Cybersecurity

Abstract

This thesis is the result of a project whose objective consists the design of an architecture for a Denial of Service attack detection system on Doftware Defined Networks (SDN) and its implementation using Big Data processing tools, semantic web technologies and Machine Learning classification techniques within the framework of the work carried out by the Intelligent Systems Group.

To carry out this project, a data storage module has been developed from which a data collection and injection module is developed in said storage system, a system for analyzing stored data in search of attacks and a system of semantization of said stored data.

A specific use case has been chosen and implemented through the use of SDN virtualization technologies, in such a way that a complete network deployment is not necessary to verify the operation of the system described above. This virtualized network is managed by an OpenDaylight controller, through which we will perform any interaction related to attack management. For the detection of these attacks we will use Machine Learning models previously learned using annotated cyberattack datasets. These models will be able to detect attack patterns in data related to network traffic, and it will report such attacks while also taking actions to block them.

The data both collected and generated will be semantized according to an ontology developed in this project, with the aim of making the data easily accessible.

Finally, a visualization system will allow the user to easily extract and visualize information about the network, as well as about attacks detected and multiple parameters related to such attacks. It will also allow the direct execution of semantic queries to extract specific data.

As a result, we will obtain a system for SDN networks that allows the detection of Denial of Service attacks, autonomous recovery from such attacks without affecting the service or other clients, and the structuring of the data through an ontology and process visualization.

Keywords: SDN, Big Data Analytics, Linked Data, SPARQL, Elastic, Opendaylight, Machine Learning, Cybersecurity

Agradecimientos

Gracias a todas las personas que me dieron la oportunidad de llegar hasta aquí. Gracias a aquellos que disponen del valor de depositar la confianza en el desconocido. Gracias.

113.02 kilómetros diaros. Pero el resultado mereció el esfuerzo.

Contents

List of Figures

Glossary

API Application Programming Interface

BGP Border Gateway Protocol

CIC Canadian Institute of Cybersecurity

CLI Command Line Interface

CSV Comma Separated Value

DARPA Defense Advanced Research Projects Agency

DOM Document Object Model

DoS Denial of Service

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDS Intrusion Detection System

ICT Information and Communication Technologies

ISCX Information Security Centre of Excellence

INDL Infrastructure and Network Description Language

LOIC Low Orbit Ion Cannon

L2-switch Layer 2 switching

MD-SAL Model-Driven Service Abstraction Layer

NAT Network Address Translation

NETCONF Network Configuration Protocol

NDL Network Description Language

NML Network Markup Language

OGF Open Grid Forum

ODL OpenDaylight

ONF Open Networking Foundation

OVSDB Open vSwitch Database Management Protocol

OWL Ontology Web Language

QoS Quality of Service

RDF Resource Description Framework

REST Representational State Transfer

SAL Service Abstraction Layer

SDN-NDL Software-Defined Networks Networking Description Language

SDN Software-defined Networking

SNMP Simple Network Management Protocol

SPARQL SPARQL Protocol and RDF Query Language

SQL Structured Query Language

STP Spanning Tree Protocol

STP Tuple Database

TCP Transmission Control Protocol

UCO Unified Cybersecurity Ontology

UDP User Datagram Protocol

UI User Interface

UML Unified Modelling Language

YANG Yet Another Next Generation

CHAPTER 1

Introduction

In this chapter, we will describe the context of the project. For this purpose, we will enumerate each part that composes the project, underlining its role in the final goal of this project. Moreover, we will describe the motivation for the development of this project and the selection of each one of the technologies used in its development. We will also detail the goals of this project and the structure of this thesis.

1.1 Context

The development of the Software Defined Network (SDN) architecture constitutes one of the most important advances in the networking field in recent years. Until now, network operators were forced to micro-manage the network, applying changes and taking monitoring actions at node level. This happened due to the fact that in non-SDN networks networking rules are not created and enforced by a single entity, but by each node separately. They may communicate with each other in order to coordinate such aspects as Quality of Service (QoS) or tunneling, but the "intelligence" of the network is held in each node separately. Furthermore, in non-SDN environments network operators are forced to use networking nodes, components and tools provided by a single company, therefore dramatically decreasing the flexibility of such networks.

The SDN paradigm allows us to avoid these difficulties. The SDN technology is based on the concept of a central network "controller", where all the decisions regarding traffic routing, network services and topology are taken. Hence, network nodes just follow rules designed and implemented by the controller. In order to manage the network, we only need to interact with the network controller instead of interacting with each node (as it was the case with legacy topologies).

Furthermore, the communication between the SDN controller and each node is based on the standardised OpenFlow protocol. This protocol organises communications between the controller and each node. Since it is standardised, it allows the simultaneous use of nodes form different suppliers, therefore dramatically increasing the flexibility of the network being managed.

One of the fields where the centralisation of the network management can have a considerable impact is the management of network cybersecurity. By having a centralised access to the entire network, we can monitor the network and act instantly in any element, mitigating or blocking attacks against our infrastructure. This dramatically increases the difficulty in attacking the network being managed. Furthermore, we can monitor specific elements of our network and "defend" other elements in an agile and flexible manner.

Another key concept in the development of this project is the Linked Data technology. The Linked Data paradigm proposes a semantic structure where definitions and relationships between data entities are represented. These relationships and definitions are predefined: hence, as new data is collected, it is "shaped" by the semantic structure that we have defined. The shaping of the data not only eases it use in future applications, but also facilitates the indexing and searching of data. Moreover, Linked Data is the basis for the

Semantic Web [1], which is considered the future of the Web.

We have also used Machine Learning techniques in order to infer predictions according to data collected through the SDN controller. Nowadays, Machine Learning techniques are being widely used in order to automatize multiple tasks, dramatically increasing the speed at which these tasks are being completed while avoiding human intervention. This can be applied to the detection of attacks in SDN environments.

Finally, we have also used Big Data tools and techniques in order to store, process and index complex data regarding the current status of networks being managed by each controller.

Combining these technologies we will be able to develop an SDN-based attack detection system which also indexes the information in Big Data tools, stores the data in Linked Data tuples and allows for the visualisation of such data in a Web interface.

1.2 Motivation

The main motivation of this project comes from the wide range of possibilities that the adoption of the SDN paradigm brings to the current management of legacy computer networks. The architecture of the SDN paradigm allows for the designing and implementation of new services that take advantage of the centralised access to any element or data regarding the current status of the network. This promotes the development of new modules that apply trending technologies to the management of computer networks. Having instant access to any part of the network also promotes the aggregated use of considerable amounts of data collected from the entire network instead of isolated units of data collected separately from each entity. As a result, we can obtain a global view regarding the current status of our network. This global view allows us to monitor not only the entire life cycle of each attack launched against services being provided in our network.

However, the flexibility brought by the SDN paradigm can be severely constrained by the limitations of human-based management. In order to take full advantage of the SDN paradigm, we will implement a machine-learning based attack detection system. This system will monitor and detect any Denial of Service (DoS) attacks against a service being provided in the SDN being monitored. Based on data representing the traffic involved in an attack to a service being provided in a sample network, a Machine Learning model will be learned. Specifically, we have selected a dataset representing a wide variety of attacks recorded and annotated in a controlled environment. Then, this model will be applied to a SDN environment built for this purpose. In this network, our model will be able of predicting the presence of a DoS attack. DoS attacks can have a considerable impact in the service industry. Some studies [2] value the direct economic impact of such attacks in an average of 1,5 million US dollars, which is a considerable amount by itself. However, this amount does not consider the highest cost sustained as a result of a DoS attack: the reputation damage. Furthermore, the impact and frequency of DoS attacks is expected to increase due to the sharp increase of attack vectors provoked by the adoption of the Internet of Things paradigm.

We can further increase the flexibility of this system by creating a Linked Data Ontology fitted for the modeling of attack-related data in a SDN scenario. This ontology will allow us to standardise data representing the scenario being monitored, hence easing its representation. The proposed ontology has been developed by using concepts and relationships from previous relationships, and this ontology itself can also be extended in order to implement new ontologies that include new scenarios, such as new attacks or hybrid networks.

The use of Big Data technologies further eases the development of this project. Thanks to the use of tools that automatise the pipelining of data from each source to a centralised indexing unit, we can easily create and execute queries in order to select very specific entries in datasets with a high amount of documents. For this purpose, we have followed a Data Lake approach, where all the data (both structured and unstructured) is stored in a single location. This results in a much simpler data managing and storing system.

Finally, we also wanted to develop a system that allows an easy exploration of the results of such monitoring. Therefore, we have developed a Web application that allows for the exploration of the current status of the network, while also allowing the writing of semantic queries in order to select specific units of information.

1.3 Project goals

The main foals of this project are:

- Create and deploy an OpenDaylight-controlled SDN network where a service is being provided,
- Monitor the network while an attack is performed against such service,
- Process the collected data and detect the presence of an attack and its main variables, as well as take blocking actions against such attack,
- Semantize such data, creating Linked Data tuples that represent the network, the attack and multiple relationships between both
- Create a Web GUI that presents the collected information

1.4 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The structure is the following:

Chapter 1 describes the context in which this thesis has been developed. Specifically, we describe the current status and role that each of the technologies have played in the planning and developing of this project.

Chapter 2 depicts the technologies used in the development and deployment of this software. We put the emphasis in those technologies and concepts which are key for the development of this system.

Chapter 3 describes the semantic model developed for the modeling of the data collected from the network. It also describes the learning dataset used to obtain the machine learning used in this project, which is also presented in this chapter.

Chapter 4 describes the overall architecture of the system. In this section we depict the flow of data among the different sub-modules that compose the architecture. We also describe each of the sub-modules and its purpose in detail.

Chapter 5 describes the system implemented and executed for the purpose of testing the concepts and architectures proposed in this project. This chapter details the implementation of each of the modules detailed in *Chapter 4*.

Chapter 6 presents the conclusions reached with the development and testing of this project. We also propose different paths for future work.

CHAPTER 2

Enabling Technologies

A wide array of technologies have allowed for the implementation of this project. Most of these technologies are based on recent innovations in the fields of Software Defined Networking and Data Pipelining; therefore, we will focus on the application of such systems in the detection and protection against cyber attacks in order to describe the context of this thesis.

Some of the key concepts in current innovation trends in the IT sector, such as Big Data analytics and Machine learning techniques, are yet to be fully implemented in Software Defined Network environments. In this section, we will also outline work done by researchers and companies alike towards the incorporation of such techniques and technologies in SDN environments.

2.1 Introduction

This project is based on the use of SDN technologies; however, Big Data analysis technologies and tools are also a key aspect of the proposed system. Therefore, we will describe the application of such techniques in Software Defined Network environments. We will also address the implementation of security systems that protect Software Defined Networks from cyber attacks.

In Section 2.2, Here we detail the use of Software Defined Network enabling technologies and systems; specifically, we will describe the use of Opendaylight in the role of controller of a Software Defined Network. We will also depict the workings of OpenFlow in its role of enabler of the SDN paradigm. Section 2.3 provides an overview of the main technologies and tools used in the processing of data in a scalable and flexible manner. Specifically, we describe the tools provided by the Elastic Stack and its potential use in Big Data related systems. We put the emphasis on the abilities of the Elastic Stack for the tasks of pipelining, storing and indexing of data. These abilities will come useful in the managing of the data collected from the Software Defined Network. Section 2.4.1 details the use of linked data technologies and platforms in order to represent data collected by the data collecting tools in this project. Then, in Section 2.6 we describe the use of Python libraries for the development of machine learning models based on annotated data. Finally, in Section 2.7, we will describe tools and techniques to perform attacks over services being provided in computer networks.

2.2 Software Defined Networks

In this section we will describe extensively the Software Defined Network paradigm and the technologies involved in its implementation. Specifically, we will focus on technologies that enable the use of SDN-adapted switches and the communication between each switch and the SDN controller, which is the central system in any Software Defined Network architecture.

Technologies such as Mininet [3] allow us to create and run accurate simulations of real SDN networks. Mininet uses Open vSwitch as a virtualization tool for implementing SDN-adapted switches. Each of these switches communicates with the controller using OpenFlow, an essential protocol in the implementation of SDN networks. This protocol organises the communication between the SDN Controller and each element of the network.

The Software Defined Network paradigm is described in Section 2.2.1. Next, the Mininet

simulation platform is discussed in Section 2.2.2 OpenFlow protocol is briefly viewed in Section 2.2.3. Then, the inner workings of the Open vSwitch tool are explained in Section 2.2.4. Finally, in Section 2.2.5 we present the Opendaylight project.

2.2.1 The SDN paradigm

The Software Defined Network paradigm has been one of the most important innovations in the computer networking field in recent years. The adoption of SDN-based architectures has dramatically increased the control that network managers have over their networks. Furthermore, the presence of a centralized point of control in the form of an SDN Controller eases the deployment and execution of networking-related services by network operators. Hence, the SDN paradigm has enjoyed widespread support from network operators such as Telefonica [4], Orange [5] and Vodafone [6].

The SDN paradigm is based on the concept of displacing the "decision-making center" in a network from each node individually to a collective module known as "SDN Controller". This controller has a global view of the network and can communicate itself with any of its elements. Since it has a global view of the status of the network, the controller can access much more information when taking decisions regarding the functioning of the network and the services provided within it. This stands out when compared to *legacy* networks, where decisions were taken by each node separately. In that case, each node can access data regarding its own status and the statuses of its neighbours, but it can not access data that portrays the status of the network as a whole or the status of the services that are being provided within that network.

Another advantage of the SDN paradigm consists in the fact that, in order to collect data or apply changes regarding multiple elements of the network, we only need to access the controller. In *legacy* networking we would have to access each network device separately, and then aggregate the information collected. This could cause issues such as the difficulty of accessing some of these elements, or scalability issues when the number of devices to be queried is too large. We would also have to develop a system tasked with the aggregation of data from multiple sources. However, in the case of SDN networks, all the information is already aggregated and stored in a single, accessible module.

In conclusion, the SDN paradigm provides multiple advantages when managing computer networks. These advantages are particularly notable when collecting and managing network data.

2.2.2 Mininet

Mininet is a virtualization system that allows us to develop and deploy virtual networks over a single machine. It is one of the main network virtualization tools used for SDN-related researching, since its switches support OpenFlow (2.2.3), the standardised protocol for communications between the controller and each node. An overview of its architecture can be seen in Figure 2.1

Among its multiple features, it provides the user with a Python-based Application Programming Interface (API) that allows the creation of tailored topologies and services. Through the use of this API we can define scenarios that mimic real-life networks and services. It also provides a Command-Line Interface (CLI) for debugging and running network-wide tests. Furthermore, since Mininet hosts run standard Linux network software, we can develop and deploy network services using them as servers. This allows us to virtualize scenarios that accurately represent the providing of a service.

Finally, due to its simplicity, it can be easily deployed in a Docker container. Hence, the scenario developed can be exported and run in any system with a Docker distribution installed.

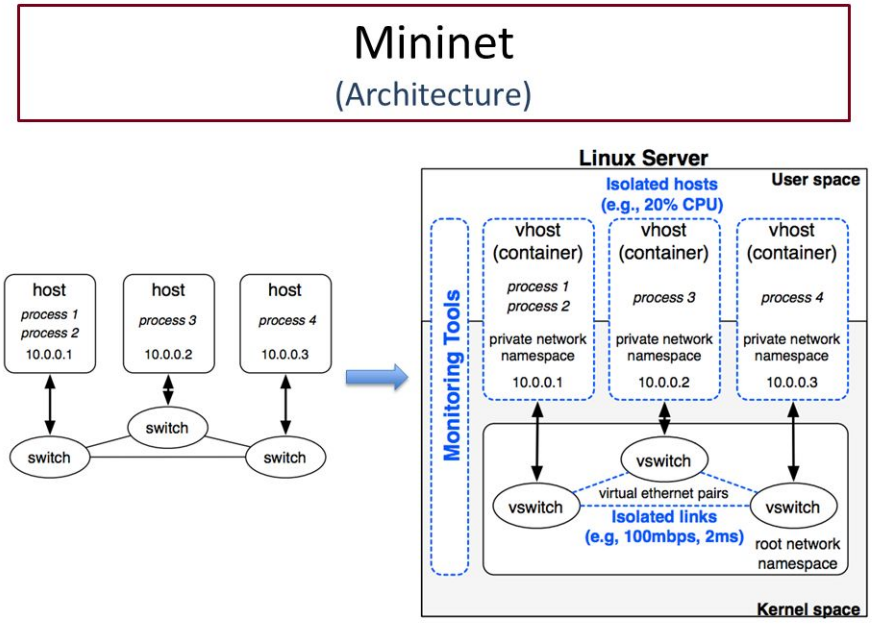

Figure 2.1: Mininet architecture

2.2.3 OpenFlow

OpenFlow [7] is a standardised communications protocol that regulates the communications between each node and the SDN controller in a SDN network. Specifically, this protocol enables the SDN controller to manage the forwarding plane of a SDN switch directly. Therefore, it is an essential element in the implementation of the SDN paradigm, since it allows the application of forwarding rules in each network node, therefore implementing the most basic service in a computer network: packet routing.

OpenFlow is considered the first fully defined protocol that tackled the lack of standardisation in the communications between each node and the controller. Since its first stable version (1.0, released on December 2009) it has been extended multiple times, adding new capabilities. Such capabilities include software-based traffic analysis, centralised control, dynamic updating of forwarding rules and flow abstraction [8]. These capabilities have contributed in turning OpenFlow into the foundation for the development of SDN-based solutions.

A typical OpenFlow solution is based on the concept of "flow". A flow defines a set of matching rules and actions; therefore, each incoming package is evaluated in order to match it to a flow entry. When a flow and a packet are matched, the actions indicated in the flow entry are taken. In each node, flow entries are stored in a flow table. This table is being monitored and managed by the SDN controller. The OpenFlow communication protocol defines and regulates the way in which the controller manages each flow table. This managing involves creating and deleting flow entries, and analysing multiple "matching" degrees when a new package enters the node.

This protocol is being currently managed by the Open Networking Foundation (ONF), which is a non-profit organisation tasked with the promotion of SDN technologies and applications.

2.2.4 Open vSwitch

Open vSwitch [9] is an open-source, virtual switch commonly used in SDN environments. It provides many features useful for the virtualisation of SDN environments, such as OpenFlow protocol support, multi-table forwarding pipelines and forwarding layer abstraction. And, as opposed to commercial solutions (such as Cisco 1000V or vCenter), Open vSwitch is designed to be controlled and managed by third party controllers; hence, it fits perfectly our needs. In order to manage Open vSwitch instances, Open vSwitch also defines the Open vSwitch Database Management Protocol (OVSDB).

It is also one of the switch virtualizers provided by default with Mininet. The combination of virtual switches provided by Open vSwitch and virtualised hosts and networking provided by Mininet allows us to deploy a fis anully operational SDN setting. By adding a SDN controller to such setting, we can reliably replicate an SDN network.

In conclusion, Open vSwitch defines and deploys the nodes that will act as switches in our virtualised network.

2.2.5 OpenDaylight

OpenDaylight [10] is an open-source, model-driven modular platform hosted by the Linux Foundation that functions as an SDN controller. Along ONOS [11], OpenDaylight is the most popular open-source SDN controller available. This can be seen in the support that such companies as Cisco, Ericsson, RedHat or ZTE provide to OpenDaylight. OpenDaylight is also promoted by network operators, such as AT&T or Comcast. The aim of OpenDaylight consists in facilitating and providing a community-led, industry-supported open source framework in order to promote and accelerate a robust SDN platform.

The core of the OpenDaylight architecture is the Model-Driven Service Abstraction Layer (MD-SAL). This layer processes each network device or application placed under and represents it as an object (or model) to upper layers. Every interaction of between these upper layers and any model is also processed within the MD-SAL. In order to represent each device or application as an object, we use models defined using Yet Another Next Generation (YANG), a data modeling language used to model configuration and state data manipulated by the Network Configuration Protocol (NETCONF) protocol. This models are configured in advance and provide generalised descriptions of the devices or applications and its capabilities located under the abstraction layer. This layer avoids the necessity of being aware of the specific implementation details of each device or application. Within a SAL, each model can have a different task. A “producer” model implements an API and provides data, while a “consumer” model consumes such data.

One of the main features of the OpenDaylight platform is its modular nature. Due to the ability of using multiple modules, OpenDaylight includes support for a broad set of SDN-related protocols (such as OpenFlow, OVSDB, NETCONF or BGP). OpenDaylight also allows the selection of multiple southbound protocol modules. These modules can be packaged together depending on the requirements of the scenario to be built.

Each component is also isolated as a Karaf feature. This packaging not only allows the user to select and install only the features needed for each specific environment; it also ensures that code added by the developer does not interfere with mature and tested code.

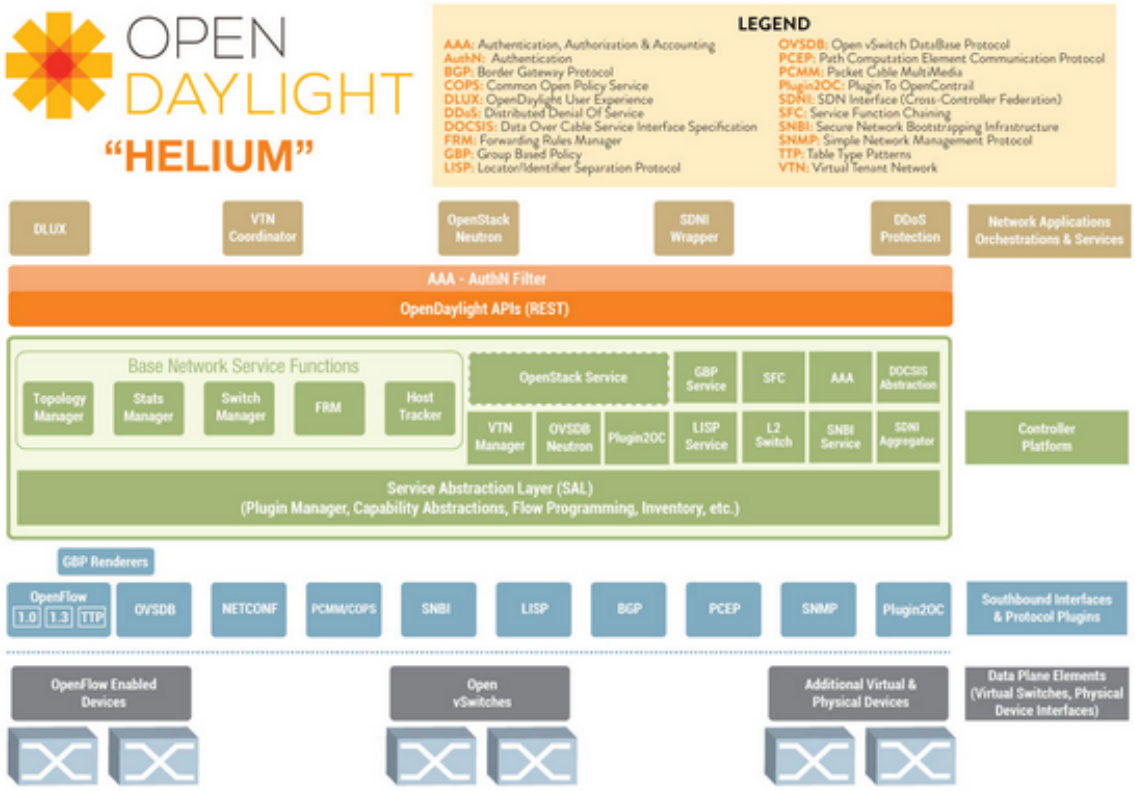

Figure 2.2: Opendaylight architecture

Figure 2.2 depicts the architecture, where the modular nature of OpenDaylight can be truly noticed. As we can see, the SAL hides a variety of modules that represent interfaces implementing multiple network protocols. Within the controller platform we can distinguish multiple services. Each one of these services is deployed as a Karaf feature that can be installed or uninstalled on demand. Some of these services are installed beforehand since they are necessary for the controller to perform its most basic tasks. The upper Representational State Transfer (REST) Interface controls the access to each of the service modules.

In conclusion, OpenDaylight provides a highly flexible and scalable SDN controlling platform able of providing multiple networking services.

2.3 Big Data Technologies

Big Data technologies are enjoying wide success in their application in many different contexts. However, the use of data in an extensive manner requires specific software. This software must be highly scalable and modular, so as to easily build data analysis systems.

One of these software systems is Elastic Stack. Elastic Stack is a set of systems developed for the processing, pipelining, indexing and visualisation of data in high quantities. ElasticSearch, which is the main indexing element of the stack, is described in Section 2.3.2, The pipelining tools Logstash and Beats are detailed in Section 2.3.3.

2.3.1 Elastic Stack

Elastic Stack is an open-source set of tools developed for the creation of data pipelines, indexing of data, and visualisation in a scalable and modular manner. It is also commonly known as the ELK Stack, since ELK is the acronym formed by the initial letters of its three main products: ElasticSearch, Logstash and Kibana. However, the Elastic Stack also contains many other tools, such as Beats or Elastic Cloud Enterprise. A small description of each module can be found next:

- ElasticSearch is the central piece in any Elastic Stack architecture. It serves as the main storing unit where the data is indexed.
- Logstash is tasked with the collection of data from different sources. It allows for the specification of multiple processing steps between the collection and storing tasks. Therefore, it is an essential tool in the deployment of data pipelines.
- Beats is a tool used for monitoring certain folders or files and sending any changes in the monitored elements to Logstash or ElasticSearch directly. It is used mainly as a data collecting tool.
- Kibana is a visualisation module adapted to ElasticSearch. It allows the exploration of high quantities of data indexed in ElasticSearch and its visualisation and querying.

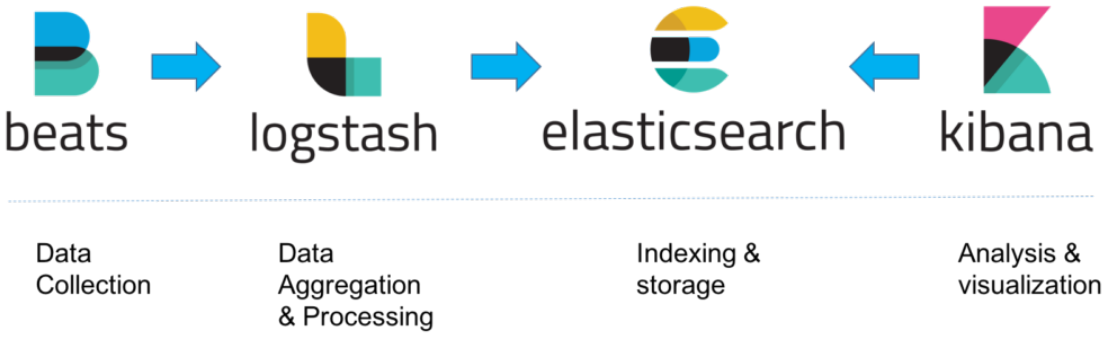

Figure 2.3: Elastic Stack Pipeline

One of the most common uses of this set of tools consists in the deployment of a data centralization and exploration system. Several instances of Beats are deployed in those systems where there is data to be monitored. Then, either a single or multiple Logstash instances receive Beats messages with new data. This data is processed and pipelined for its indexation in an ElasticSearch instance. Then, a Kibana instance queies the ElasticSearch module in order to display the data in many different forms. This process can be seen in Figure 2.3.

In the following sections each ElasticStack module will be described in more detail.

2.3.2 ElasticSearch

ElasticSearch [12] is a module that contains a distributed, RESTful indexing and analytics engine which is capable of solving a growing number of use cases. It is key to the ElasticStack architecture, since it takes on the task of storing the data in a centralized and scalable manner.

ElasticSearch represents data as schema-free JSON documents, where each document corresponds to a JSON document representing each entry of the data indexed within the ElasticSearch instance. Furthermore, an ElasticSearch instance can hold multiple indices, and each index can hold many different types of documents. ElasticSearch also provides a primitive Hypertext Transfer Protocol (HTTP) web interface where the results of queries can be consulted. We can also obtain statistics regarding each of the indices, such as the amount of documents or its size.

ElasticSearch's distributed nature allows it to gain speed and stability from each additional node. Due to this distributed nature, documents stored in ElasticSearch are partitioned into different containers or shards, which can be stored on a single or multiple nodes. We can duplicate these shards to provide redundancy, or we can balance access to the data between these shards.

2.3.2.1 Lucene

The core of the ElasticSearch indexing and querying function is based on Lucene. Lucene [13] is an open-source Java library that implements common search and searching-related tasks such as indexing, querying, highlighting and many others. ElasticSearch can be seen as a Lucene "wrapper", allowing the user to use Lucene as a server-side search application and automating such tasks as index management and distributed coordination.

Lucene works by adding content to a full-text index. It then allows the user to execute

queries on this index, returning documents sorted by following a criteria specified in the query. If no criteria was specified, Lucene finds the relevance of each document to the query being executed. Lucene achieves fast responses since it searches indices instead of the text directly.

2.3.2.2 Adding and searching data

ElasticSearch provides different means to upload data. We can configure other ElasticStack tools, such as Logstash and Beats, in order to send data directly to multiple indices within the ElasticSearch instance. We can also create new indices containing the data being sent by such modules.

However, we can also add data to the ElasticSearch instance without using any other ElasticStack module. For example, we can use HTTP requests sent directly to the Elastic-Search index. An example of such message can be seen in the following listing:

```
PUT /website/blog/123
{
  "title": "Elastic is funny",
  "tag": [
    "lucene"
  ]
}
```

In this message, we are creating a new sample document and storing it in the "website" index. The document created is of type "blog", and its id is "123".

We can also use the ElasticSearch API libraries (available for Java, JavaScript, Go, .NET, PHP, Perl, Python and Ruby) to manage data being stored in ElasticSearch. The use of this APIs allows us to seamlessly integrate ElasticSearch into any application being developed using any of the languages being supported by ElasticSearch APIs.

In order to search data in ElasticSearch, we need to compose queries that represent the criteria selected for searching such data. Once we have translated our searching requirements into a query or set of queries, the ElasticSearch engine will return a set of documents that match our criteria. This searching action can be performed either through HTTP requests or by using any of the API libraries mentioned before.

2.3.3 Beats

Beats is a data monitoring ecosystem that allows us to automate the parsing and collecting of data. It acts as an agent that can be deployed in any machine and is tasked with the monitoring of any type of file or folder, sending a message (known as "beat") for each new entry in such file or folder. Beats accepts many types of log formats from such widespread systems as MySQL, Apache, NGINX and more. However, Beats can also monitor and collect raw data.

Once the data has been collected by Beats, it can be directly sent to the ElasticSearch instance. However, the most common practice consists in deploying a Logstash instance that performs processing and pipelining tasks. An example of the role of Beats in the Elastic Stack scheme can be seen in Figure 2.4.

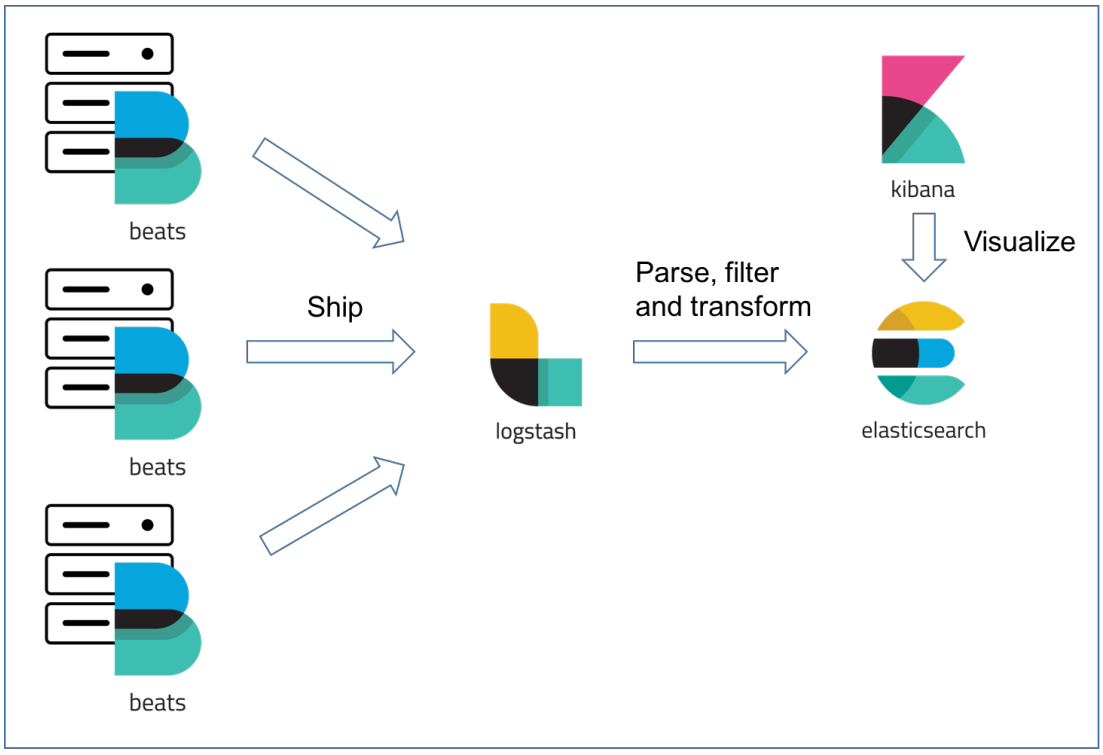

Figure 2.4: Beats in the Elastic Stack scheme.

2.3.4 Logstash

Logstash allows us to ingest data from multiple sources simultaneously, therefore significantly easing the aggregation of data.

Logstash is controlled by a configuration file in which the inputs, outputs and data pipeline steps are specified. Logstash allows us to define filters that parse data being received. It also allows us to specify conditional actions, therefore performing different processing steps depending on the data being received in the Logstash instance. The fact that we can configure Logstash to execute certain scripts depending on certain conditions given by received data and internal variables allows us to integrate seamlessly Logstash instances with current applications.

Finally, Logstash can route events to other output plugins which can forward these events to other external programs besides ElasticSearch. The role of Logstash in the Elastic Stack can also be seen in Figure 2.4.

2.4 Semantic Technologies

Semantic technologies are based on a single concept: the use of formal semantics to give unstructured data a meaning. The use of semantic technologies lets us build concepts and relationships between data with completely different formats and sources. This key definition is the origin of the term "Linked Data". Furthermore, in order to implement these universal concepts and relationships between data, we need to define a semantic-oriented standard. The most widespread semantic-related standard at the moment is the Resource Description Framework (RDF), which is used to write data schemes which shape the semantization of data. In order to execute queries over semantized data, the SPARQL querying language is defined.

In the following sections, these concepts will be further explained. The Linked Data paradigm will be described in Section 2.4.1. Next, in , the RDF standard will be introduced. Lastly, in section 2.4.2 the SPARQL language will be outlined.

2.4.1 Linked Data

Linked Data [14] is a paradigm that represents the idea of publishing data in a structured manner, using predefined vocabularies, entities and relationships. These features allow the data to be connected to other data instances. They also allow for the data to be interpreted by machines. Linked Data is defined by four principles:

- Use URIs to identify data entities
- Use HTTP URIs to publish these entities

- Annotate these entities, so when someone looks up a URI we can provide useful information
- Include links to other entities represented by other URIs.

In order to structure the data, first we have to provide predefined structures. These structures are specified using the RDF standard. Defining models that follow the RDF standard allows us to create interchangeable data structures, which facilitate data merging both between similar and different schemas. These structures are based on the concept of "triple". A triple is a data instance composed by three entities: a "subject", a "predicate" and an "object". This structure follows the classical entity-attribute-value commonly found in object-oriented programming. When we put together many triples, we obtain a RDF graph. This graph can resemble a connected network of nodes, since an object in a triple can be a subject in another triple. The common interaction between this graph and other systems can be seen in Figure 2.5.

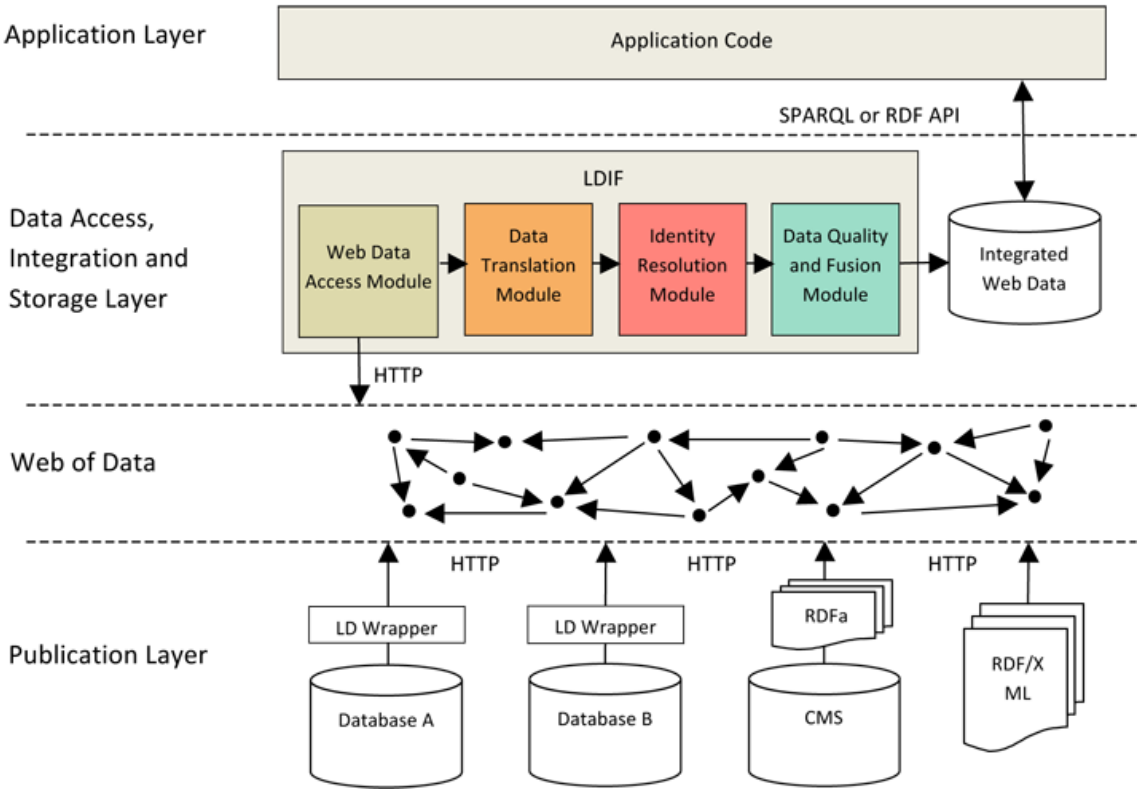

Figure 2.5: Integration of Linked Data schemes

In order to write out these graphs, several languages can be used. The RDF/XML language is the most common option used for developing files representing RDF graphs. However, there are other possible choices, such as Notation3 [15]. Notation3 is a serialising

language developed to increase human readability of RDF graphs. Yet another option is Turtle [16]. Turtle is a RDF syntax similar to the one provided by SPARQL. Finally, another common option to formalise RDF graphs in files is JSON-LD [17]. JSON-LD is a language based on the features provided by the JSON data representation language. This is the format which is going to be used in this project.

RDF is based on a set of classes grouped in the RDF Schema. RDF Schema is an extension of the basic RDF vocabulary which provides multiple tools in order to allow the creation of new RDF vocabularies. It defines such mechanisms as "Class", "domain" or "range", which allow us to create new vocabularies that make use of such definitions and relationships. These new vocabularies are said to be "RDF-Schema defined vocabularies".

We can also use Ontology Web Language (OWL) [18] to further define the semantics of the extended vocabularies that we have created. The use of OWL allows us to define further relationships between instances or infer implicit facts.

Once we have used RDF tools in order to define data, we obtain triples representing instances stored in the data and relationships between each instances and these instances and their attributes. However, we have to store these triples in systems that provide searching and indexing services adapted to these triple-based structures. One of the most common engines used for storing and indexing triples is Apache Jena Fuseki [19]. Jena Fuseki functions as a SPARQL server with a storage layer. It stores triples and allows the user to execute SPARQL queries in order to collect data.

SPARQL is a SQL-derived querying language specifically designed for RDF-based data. In the following section we will further describe this language.

2.4.2 SPARQL

SPARQL [20] is a querying language specifically designed to perform queries over databases storing RDF data. SPARQL was developed as a language inspired in the SQL querying language for regular, relational databases. Therefore, it shares with SQL much of its syntax structure. On 15 January 2008, SPARQL became an official W3C recommendation for its use as a querying language in the semantic web.

As we stated before, Jena Fuseki provides the user with a SPARQL editor where data queries can be written and executed. Here we can see an example of a SPARQL query:

```
PREFIX ex: <http://example.com/exampleOntology#>
SELECT ?capital
       ?country
WHERE
```

```
{
  ?x  ex:cityname       ?capital   ;
      ex:isCapitalOf    ?y         .
  ?y  ex:countryname    ?country   ;
      ex:isInContinent  ex:Africa  .
}
```

In this query we are selecting all the country capitals in Africa.

2.5 Data Visualization Technologies

2.5.1 Visualization libraries

D3.js [21] is a JavaScript library that allows us to bind our data to a Document Object Model (DOM). Once this data is binded, we can apply data-driven transformations to the document being displayed. This includes creating and displaying HTML elements that depend on the value of certain data fields, such as tables of charts. We can also create elements that allow the interaction with such data.

D3 puts the emphasis on complying with web standards for the purpose of providing access to all the capabilities of multiple modern browsers. This allows the user to avoid tying him/herself to the use of a proprietary framework, combining powerful visualisation components with a data-driven approach to DOM modifying.

Instead of focusing of providing every possible feature available, D3 focuses on manipulate efficiently the data-based documents. This design decision enables D3 to support large-scale datasets and dynamic behaviour, allowing a fast response to changes in the environment. D3 also enjoys a widespread community support, which provides a wide array of templates of HTML elements driven by the presence of different types of data and its changes. These features are particularly interesting for our project, since our intention consists in representing data via HTML elements that change as data changes.

2.5.2 Sefarad

Sefarad is a web application developed for the purpose of exporing semantic data by implementing and executing SPAeRQL queries to predefined endpoints without coding this behavior. This system has been developed by the Intelligent Systems Group (GSI) at the Technical University of Madrid (UPM). It allows the user to easily design a graphical interface for the purpose of presenting Linked Data by combining a set of data-driven visual

plugins that represent the data stored in a SPARQL endpoint. Hence, linked data can be easily explored.

Sefarad is based on Web components. Web Components are a set of standards currently being produced by Google engineers as a W3C specification that enables the creation of reusable widgets or components in web documents and web applications. The intention behind them is to bring component-based software engineering to the World Wide Web. The component model enables encapsulation and interoperability of individual HTML elements.

2.6 Machine Learning

Machine learning techniques are being widely used for the automation of problems resolution within multiple sectors, such as computer vision, variable prediction and natural language processing. In this section, we will review the main tools used for the implementation of machine learning techniques within the scope of this project.

Specifically, we will describe the main library used for the learning and implementation of machine learning models in Python, which is Scikit-learn. Scikit-learn [22] is an open source machine learning library for the Python programming language. It enjoys widespread support as one of the most praised Machine Learning libraries currently available in the open source community, and it is being used in most Python-based Machine Learning applications being deployed today. Its API-centered design sharply increases its usability in end-to-end Machine Learning projects in the research, development and production phases. This library is designed to interact with SciPy and NumPy [23], two of the most widely used calculus libraries in Python, as it was originally created as an extension to the already-existent SciPy library.

Scikit-learn provides the user with a broad set of Machine Learning methodologies for the resolution of both supervised and unsupervised machine learning tasks. Scikit-learn provides tools and methods to develop such techniques as Clustering, Ensemble-based learning, Nearest Neighbours, Multilayer Perceptrons, Support Vector Machines or Decision Trees. Some of these techniques will be particularly useful in the development of a solution for the analisis being performed in this project. We can see an overview of the algorithms implemented by Scikit-learn in Figure 2.6.

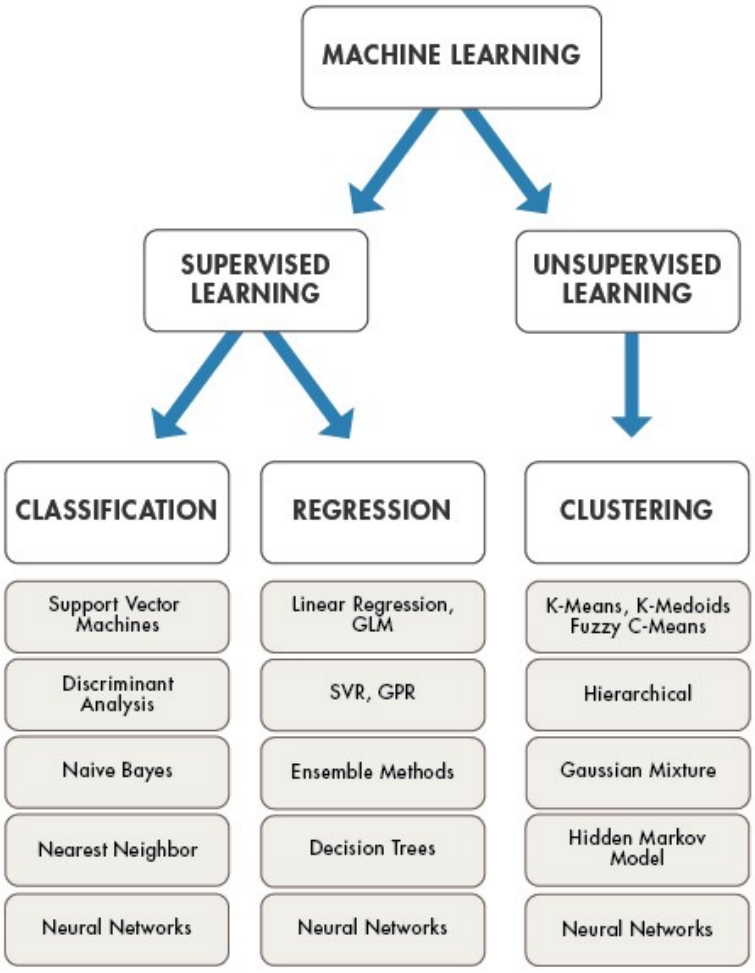

Figure 2.6: Categories of Machine Learning algorithms

2.7 Cyber-attack datasets and tools

In this section, we will describe the attack techniques and tools used to replicate the situations described in the datasets used in this project. These techniques and tools will allow us to replicate the annotated datasets used to learn the Machine Learning models deployed in this project. Specifically, we will describe the attack technique chosen and the tool selected for such purpose. We will also introduce the data processing tool used in the process of detecting attack patterns in the data collected.

2.7.1 CSE-CIC-IDS2018

The Communcations Security Establishment - Canadian Institute for Cybersecurity - Intrusion Detection System 2018 dataset [24] is one of the most complete annotated datasets

currently available with an open license. It consist in 1,4 GB of logs representing many flow metrics collected at network level and annotated with attacks provoked while those measurements were being taken. Due to this, the dataset presents an fitting structure in order to perform Machine Learning techniques to detect patterns that might outstand over the rest of the data. We could use the detection of these patterns to find relationships between values in each metric and the label that indicates if an attack is being suffered by a service being provided in such network. This network can be seen in Figure 2.7.

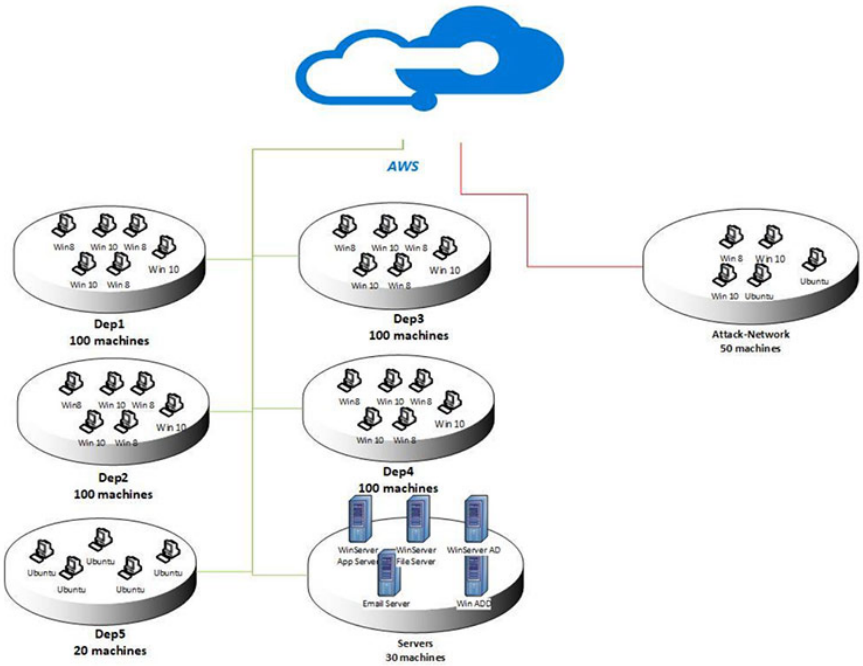

Figure 2.7: Topology of the CIC network

Even though many attacks have been simulated in this network, we are specifically interested in the detection of DoS attacks. For this purpose, we have analysed the tool used for the execution of DoS attacks within the environment described by the CSE-CIC dataset. This tool is described in Subsection 2.7.2. We will also describe the tool used for the collecting of the metrics related to each of the services being provided. This tool is portrayed in Subsection 2.7.3

2.7.2 Low-Orbit Ion Cannon

One of the attacks represented in the dataset provided by the Canadian Institute of Cybersecurity consists in a HTTP-based denial of service attack. The concept being implemented in this attack is based in establishing as many Transmission Control Protocol (TCP) con-

nections as the service being attacked can possibly control. Therefore, once the socket limit of the process controlling the service is reached, the service remains unusable for any other user that may try to access such service.

There are several tools and methods of implementing this type of attack. Tools such as Slowloris or HOIC allow us to easily prepare and execute DoS attacks without relying on huge amounts of traffic that could be easily detected. However, in the case of the scenario being represented by the selected dataset, the tool used for executing DoS attacks is the Low Orbit Ion Cannon (LOIC).

The Low Orbit Ion Cannon is one of the most used DoS applications. It is an open source application developed for Windows OS by Praetox Technologies. This application allows the user to configure a TCP or User Datagram Protocol (UDP)-based attack. In the case of TCP attacks, a configurable number of connections are opened and kept open for the purpose of occupying all the available sockets in the servers providing the attacked service.

2.7.3 CICFlowMeter

CICFlowMeter [25] is a network traffic flow analyser used by the providers of the CSE-CIC dataset in order to build the annotated datasets used in this project. CICFlowMeter takes traffic capture files as an input and generates metrics regarding Bidirectional Flows, where the first packet of each flow determines the forward and backward directions. Depending on the flow representing a TCP connection or an UDP connection, the condition for the closing of each flow is given by the detection of FIN packets (in the case of TCP connections) or a timeout (in the case of UDP connections).

CHAPTER 3

Cyber-attack Detection Models

Semantic technologies are being swiftly adopted in many areas of the Information and Communication Technologies (ICT) industry. Furthermore, they have been widely embraced in web-related technologies, resulting in the creation of the concept "Semantic Web", which enjoys a broad success among service providers. However, the use of semantic models is more sparse in other areas, such as SDN. More so, there is a lack of semantic models that represents SDN-related data within the context of a cyberattack. Due to this, we have proposed an extension to present semantic models that allows us to represent the occurrence of attacks targeted at services being provided in our network.

Machine Learning technologies are also being widely adopted in the field of attacks detection. The use of Machine Learning techniques allows us to fully automatise the whole lifecycle of the attack, from its detection to the taking of protective measures. Thus, in this section we will present a Machine Learning model learned from the data provided by the Canadian Institute for Cybersecurity that will be later integrated with other subsystems for achieving full protection against DoS cyberattacks.

3.1 Introduction

The aim of this chapter consists in introducing and describing the semantic and Machine Learning models designed and learned for this project. First we are going to present the semantic model used as a basis for the development of our cybersecurity-focused SDN semantic model. This model was selected due to the fact that it takes into consideration multiple metrics that are specifically adapted to the data structure of OpenDaylight-controlled SDNs. Hence, it constitutes a great basis for the development of a security-focused SDN semantic model.

When analysing scenarios where DoS techniques are used to attack service provisioning in SDN environments, there are two main schools of thought regarding the methods and targets for such attacks. The first scenario consists in focusing on attacking the nodes that compose the network [26]. Specifically, due to the specification of the OpenFlow protocol, when an OpenFlow node receives a new packet which does not match with any rules being currently enforced, the node stores that packet and sends an OFPT_PACKET_IN message to the controller. However, if the node does not have any space left in its buffer, it sends the entire packet along the OFPT_PACKET_IN message to the controller. In this case, the controller responds with an OFPT_PACKET_OUT message that also includes the entire packet. This behaviour can be exploited by saturating a node with unmatchable packets in order to fill the buffer and saturate the links between each node and the controller by forcing the node to send and receive from the controller the whole data packet.

Another approach to SDN-based DoS attacks would be the targeting of a specific service being provided through a SDN network. In this case, the attack resembles a server-targeted attack in a legacy network. However, the use of SDN allows us to monitor multiple interfaces in a centralised manner and taking synchronised countermeasuring actions in multiple nodes throughout the network. This approach will be followed during the development of this project.

Once we have selected an approach, we will train and develop a machine learning model tasked with the detection of flow patterns that are related to attacks being executed against services provided by the SDN. We will test the use of multiple Machine Learning techniques in order to select the algorithm with the highest possible F1-Score.

This chapter is divided in two main sections. In Section 3.2 we describe the semantic models used as basis in order to build our extension focused on attack scenarios over SDN environments. Then, in Section 3.3 we describe the steps followed in order to test multiple Machine Learning algorithms and select the most fitting one according to multiple selection

criteria.

3.2 Semantic modeling

The model used in this project has been developed by merging and extending other models especialised in SDN networks and cyberattacks. In this section we will describe the process followed in the development of this semantic model.

In Subsection 3.2.1 we describe the semantic model used as basis to represent data collected from the controller regarding the general status of the network. Next, in Subsection 3.2.2 we describe the ontology used as basis for the inclusion of cyberattacks concepts and relations in the previous model. Finally, in Subsection 3.2.3, we describe the model resulted from the combination and extension of both previous models.

3.2.1 SDN-NDL

The SDN-NDL ontology [27] was developed with the goal of implementing a framework for autonomous fault management in SDN environments. Therefore, it contains many classes and relationships that represent metrics and variables regarding the functioning of a SDN network. Hence, we have selected this ontology as the base of our model, since it allows us to model the data directly collected from the SDN controller.

In SDN-NDL the representation of the topology of a SDN network is centered around the **Node** class. This class represents each of the switching devices present in any network. Specifically, the **openflowNode** class represents the variables and features provided by the nodes that implement the OpenFlow protocol (as is the case with SDN switches). The entity hierarchy associated to that element can be seen in Figure 3.1. As we can see, each openflowNode entity has a set of interfaces (represented in the ontology by the class **Interface**) that connects it with other elements of the network. Following the OpenFlow protocol specifications, each openflowNode has a Flow Table where it stores the rules being sent by the controller. In this ontology, the **FlowTable** class represents such table; furthermore, those rules are structure into instances of the **Flow** class, which are stored in the **FlowTable**. The ontology also allows for the monitoring of network statistics aggregated by tables through the use of the **FlowTableStats** class.

Furthermore, the **Flow** class includes the **FlowStats**, **Instruction** and **Match** entities, as we can see in Figure 3.2. The **Instruction** class, which can be seen in further detail in Figure, represents the different actions that can be specified within a flow of an OpenFlow

node configured by a SDN controller. In fact, the SDN-NDL ontology defines an **Action** class that encompasses these actions. Regarding the **Match** class, this entity represents the matching rules that are specified in each flow rule in order to assign a behaviour to each traffic flow.

Specifically, the actions considered in this ontology are three: **dropAction**, **groupAction** and **outputAction**. This array of actions accurately represent the most common actions being performed by nodes in a SDN network. Regarding matching rules, we can specify matching rules according to Ethernet address (**ethernetMatch**), input or output ports (**inPort**, **outPort**), or vlan tags (**vlanMatch**).

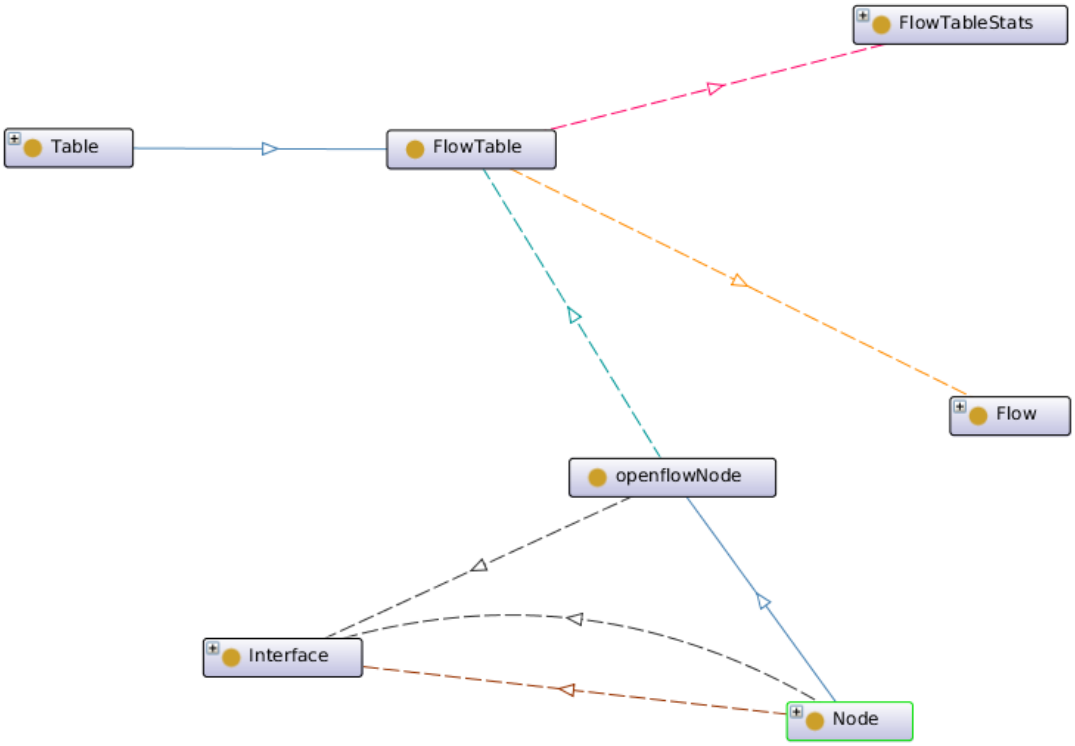

Figure 3.1: Class Hierarchy for the openflowNode entity

The **FlowTableStats** class aggregates just one of the multiple types of statistic data that is available. We can also monitor statistics from each flow separately and statistics regarding the traffic flow in each interface. Thus, we define two new statistics-related classes: **InterfaceStats** and **FlowStats**. These entities, along with **FlowTableStats**, are subclasses of the **NetworkStats** entity, which is directly controlled by a service represented in the ontology by the **StatsManager** class. Finally, both **NetworkObject** and **NodeObject** have **NetworkStats** instances. This class hierarchy can be seen in Figure 3.3.

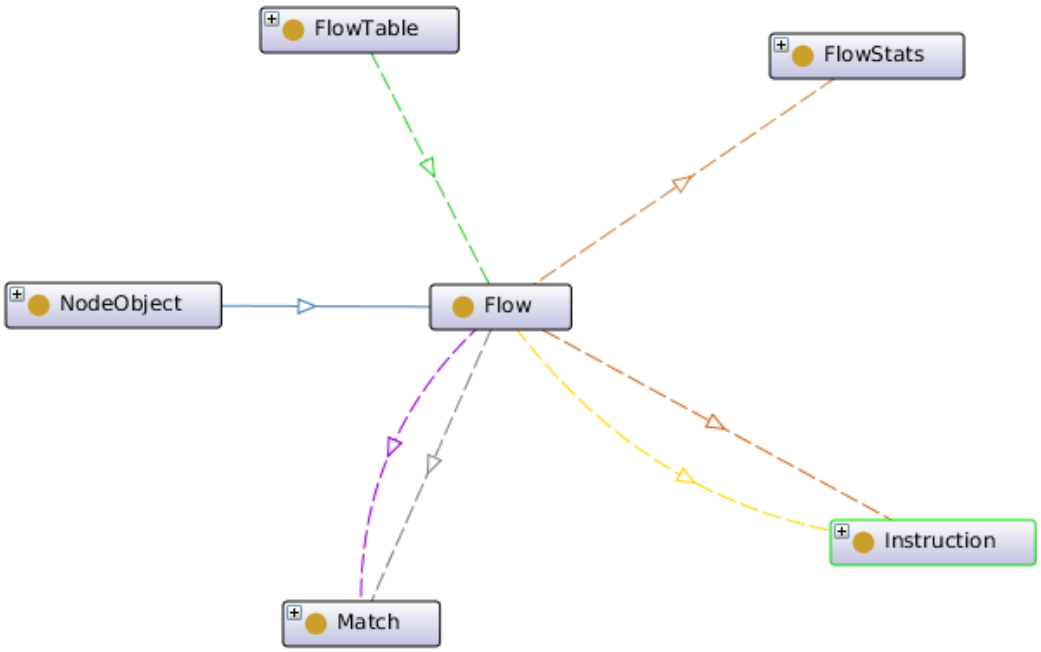

Figure 3.2: Class Hierarchy for the Flow entity

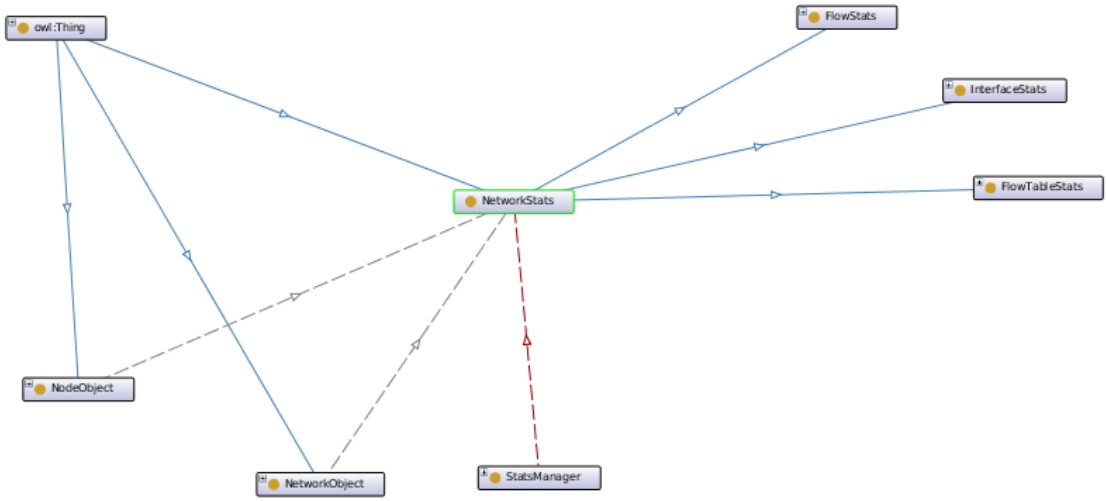

Figure 3.3: Class Hierarchy for the NetworkStats entity

These classes and relationships show that the SDN-NDL ontology is well adapted to SDN scenarios and environments. However, not only this ontology fits the SDN paradigm, but it also represents elements associated with the use of the OpenDaylight SDN controller

specifically. In order to keep track of the elements that compose the topology and its inner workings OpenDaylight use a series of self-contained services, following the OpenDaylight's philosophy of a modular architecture. These services are also represented in the SDN-NDL ontology through such classes as **HostTrackerService**, **InventoryManager** or **TopologyManager**.

SDN-NDL also includes classes and elements that allow us to represent topologies where different technologies to the ones mentioned in Section 2.2 are defined. For example, instead of using OpenFlow, the OpenDaylight controller can also use NetConf [28] to configure the flow tables of each node in the topology. In that case, we can use the **netconfNode** entity to represent such nodes.

3.2.2 Unified Cybersecurity Ontology

The Unified Cybersecurity Ontology (UCO) [29] is an ontology created with the purpose of supporting the integration of cybersecurity-related information in systems that were not designed for that specific purpose. With this aim, it incorporates and integrates heterogeneous data and knowledge schemas from different cybersecurity systems while reducing them to the least common multiple. This ontology is itself an extension to the Intrusion Detection System (IDS) ontology [30], extended with the purpose of including other types of attacks instead of just network intrusion attacks and events.

This ontology is built upon the **UCOThing** class, which serves as the "parent" class that every other entity in the topology inherits. Then, a series of classes which represent a typical attack scenario are defined. We are going to describe the ones that we have considered to be the most important for our use case.

Since the UCO ontology has been created with the aim of standardising the representation of attack-related data, one of the most important entities is the **Attack** class. The **Attack** class is the junction point for many entities that represent the means, effects and causes of attacks being executed against the network or its services. The structure surrounding this class can be seen in Figure 3.4.

The UCO ontology defines relationships between the **Attack** class and the **Observable** and **Indicator** classes, tasked with showing the visible effects of the attack in the services provided within the network. An **Attacker** class is also defined in order to describe the known parameters of the topological source of the attack.

A **CourseofAction** entity is also defined. In this class, we represent the countermeasures being taken to protect certain services or the whole network from the attack being

executed. However, the UCO ontology does not specify a set of possible actions to be taken in order to implement such Course of Action. This is only logical, since the original goal of this ontology was for it to be used in any scenario. Hence, specifying a set of possible countermeasures would limit the range in which this ontology could be used.

Therefore, one of the tasks to be completed during the extension of the SDN-NDL ontology with the UCO ontology will consist in "filling the gaps" left by the designers of this ontology with concepts from the SDN-NDL ontology in order to adapt it to a SDN scenario.

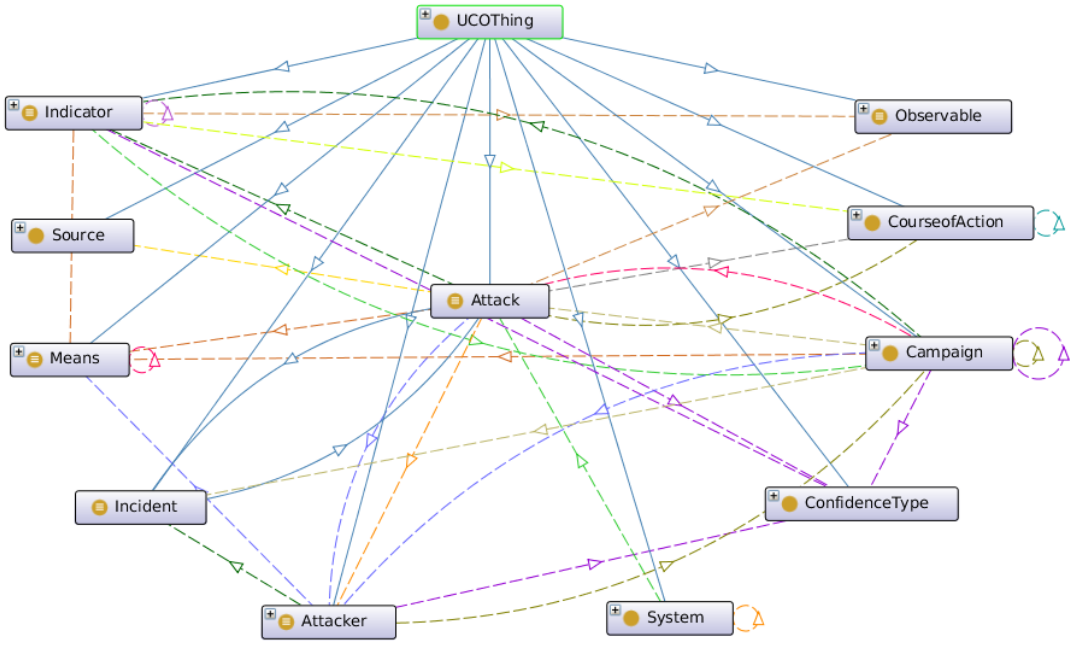

Figure 3.4: Class Hierarchy for the Attack entity

3.2.3 SDN-AR: Software-Defined Network Attack Reporting Language

In this subsection we will describe the semantic model resulted from the extension of the SDN-NDL ontology described in Subsection 3.2.1 with concepts, relationships and properties specified in the UCO ontology specified in Subsection 3.2.2. The goal of this extension consists in adapting the SDN-NDL ontology with the purpose of enabling it to represent attack scenarios over SDN networks while retaining its ability to accurately represent the correct functioning of an SDN and its statistics.

Therefore, using the SDN-NDL ontology as base, we have extended it including concepts

and relationships from the UCO ontology. Specifically, we have included the Attack concept, and some concepts and relationships related to this entity, as we can see in Figure 3.5. In this image we can see the merging of UCO concepts (such as Attack, CourseofAction or Attacker) with the SDN-NDL ontology and its concepts (HostNode, Node, Flow). In the case of CourseofAction, in a SDN environment, all actions can be taken via the use of the northbound API. Specifically, actions that affect networking rules can be translated into Flow rules and pushed into the Flow Table of any network node via the API independently of its position on the topology.

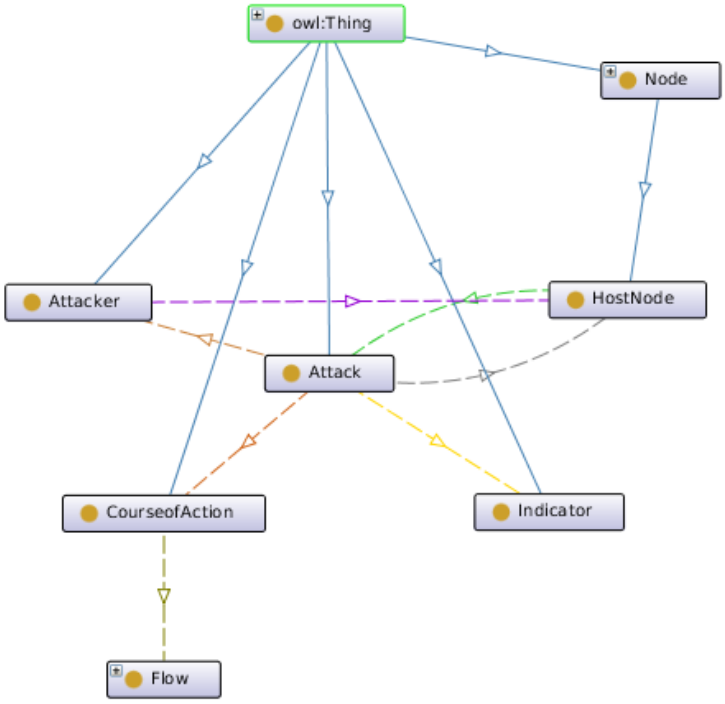

Figure 3.5: Class Hierarchy for the Attack entity within the SDN-AR ontology

Regarding attackers, we assume that attackers are hosted by Host elements in the SDN environment. Therefore, we have created a *hostedBy* relationship, which relates an specific attacker with the network element which is hosting it. This network element is a HostNode, which inherits relationships and attributes from the Node class. Finally, we have also added the Indicator class, which represents the parameters detected by the monitoring system that were used in order to infer the presence of an attack to a service being provided within the network.

Finally, we can obtain a general view from the architecture at Figure 3.6. As we can see, it consists of a combination of SDN-NDL and UCO hierarchies with the addition of

the Monitor class, which represents the monitoring system developed as a subsystem in this project.

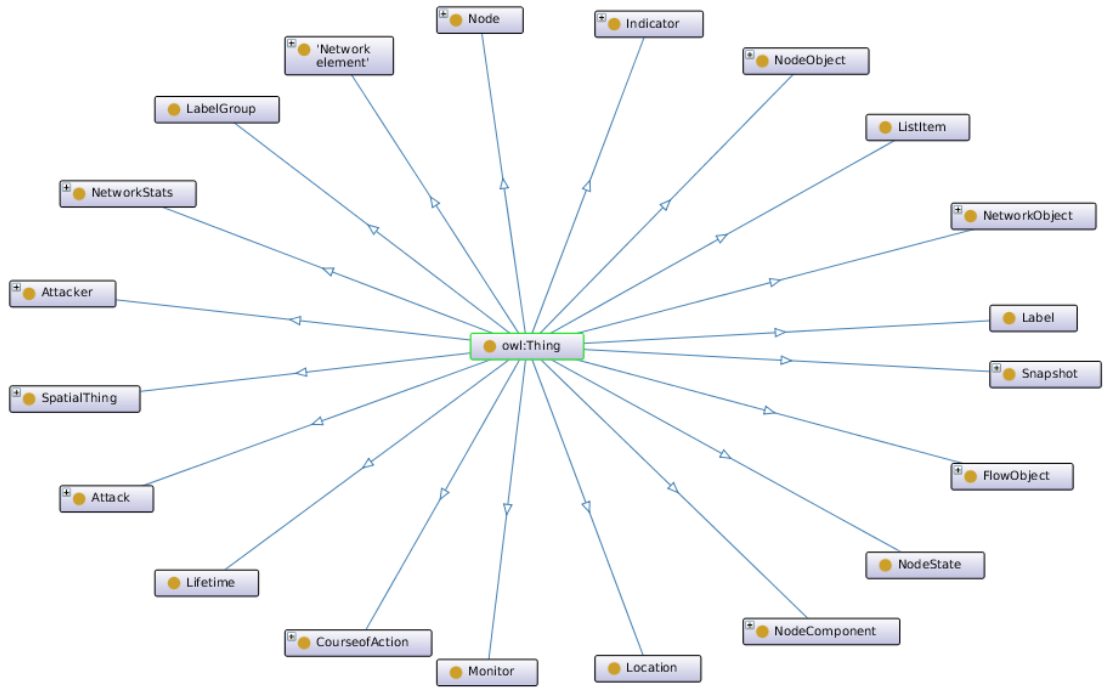

Figure 3.6: Class Hierarchy of the main classes of the SDN-AR ontology

3.3 Machine Learning modeling for cyber-attack scenarios

The learning of an accurate Machine Learning model is key, since it constitutes the core of the system that predicts the presence of an attack. Therefore, the training dataset must be carefully chosen. Once we have chosen an specific dataset to work with, we have to test multiple algorithms and configurations to select the one with the highest performance. These steps will be described in this section.

First, in Subsection 3.3.1 we will describe the dataset selected and the reasons behind the selection of this specific dataset over other possibilities. Then, in Subsection 3.3.2 we will analyse the main features of such dataset and we will portray the accuracy obtained using different algorithms.

3.3.1 Intrusion Detection Evaluation Dataset (CICIDS2017)

When facing prediction challenges two approaches can be followed: supervised learning and unsupervised learning. Unsupervised learning consists in using Machine Learning techniques in order to detect patterns and clusters in features in order to divide the data entries among different groups, since the data itself is not labelled. On the contrary, supervised learning is used when the training data is already labelled and the learning task consists in finding patterns that relate certain combination of values from certain features with each of the values of the "target" variable (the variable to be predicted).

Our focus will be put in using supervised learning techniques in order to develop the Machine Learning model; therefore, we need to find a network-based cyberattack dataset that contains annotated data regarding the attack represented by each entry. This requirement itself is a challenge. There is a considerable amount of datasets available online that store data collected during the execution of multiple types of attacks over a variety of networks. However, most of these datasets are not annotated, since it is a complex task due to the dimensionality of the data.

Regarding annotated datasets, one of the first datasets published tackling this issue was the 1999 Defense Advanced Research Projects Agency (DARPA) dataset [31]. This dataset was obtained from an ad-hoc testbed created in order to simulate a scenario with background traffic similar to that on a government site with hundreds of users. In this contexts, 58 different attack types were launched. Even though this dataset has been the main benchmark for the testing of Intrusion Detection Systems during many years [32], there are several reasons in order to discard its use in this project. First of all, since details regarding the traffic generation software used in the testbed was never made public, some authors have raised questions regarding the accuracy of such background traffic. Furthermore, this dataset was generated using tools that may be outdated.

Further advancing along the timeline of IDS-related dataset publications, we find the first contribution by the Information Security Centre of Excellence (ISCX). The 2012 ISCX intrusion detection evaluation dataset [33] specifies accurately the tools and techniques used to create the background traffic that surrounds the attacks being executed in the network. Furthermore, the dataset is divided in seven days of normal network activity with multiple intrusion scenarios specified and labelled in each dataset. Due to this features, this dataset was originally considered to serve as the basis for our learning process.

However, the ISCX (and its successor, the Canadian Institute of Cybersecurity (CIC)) has been publishing new IDS datasets ever since. Therefore, we have selected a more recent version of the IDS published by them: specifically, we have used the 2017 IDS CIC dataset

(also known as CICIDS2017) [24]. This dataset shares the data structure with previous datasets from the same research group, but it uses updated attack tools and it is based on the 2016 McAfee report on the most common attacks on current networks.

In order to create this dataset, realistic background traffic has been created following the B-Profile system [24]. B-profiling consists in individually extracting features of each flow, such as packet sizes depending on protocol, patterns in the payload, size of payload or request time distribution of protocols. Once these flows are individually profiled, they are clustered and aggregated to generate groups of users with similar flow behaviour.

In this dataset 25 users have been specifically "B-profiled" based on their HTTP, HTTP-Secured, FTP, SSH and e-mail protocol flows. These users, workers in the CIC office, have been monitored during five days. During these days, multiple Brute Force FTP, Brute Force SSH, DoS, Heartbleed, Web Attack, Infiltration, Botnet and DDoS attacks were executed. The resulting dataset was evaluated in order to check that it covers the following criteria:

- Complete network configuration: the dataset reflects data obtained from a network with a complete topology that includes a Modem, Firewall, Switches, Routers and multiple operating systems

- Complete traffic: the dataset reflects traffic created by user-profiling agents in multiple machines and real attacks

- Labelled dataset: the data is labelled in order to allow its use for the finding of patterns that establish relationships between the presence of attacks and certain flow profiles

- Complete interaction: the dataset reflects both Local Area Network (LAN) communications and Internet communications

- Complete capture: since the method used for monitoring is based on Port Mirroring, all traffic flows have been captured and recorded on the storage server

- Available protocols: the data does not represent traffic associated to a single protocol, but it monitors flows following multiple common protocols

- Feature set: more than 80 network flow features have been extracted from the network traffic files using the CICFlowMeter tool [25]

3.3.2 Classification algorithms and results

The first step in obtaining a prediction model for a certain variable using pre-determined datasets consists in analysing such dataset. Hence, our first task will be obtaining a list indexing the correlation of each feature with the "Label" feature, which is the variable to be predicted. A graph representing the results of such table can be seen in Figure 3.7

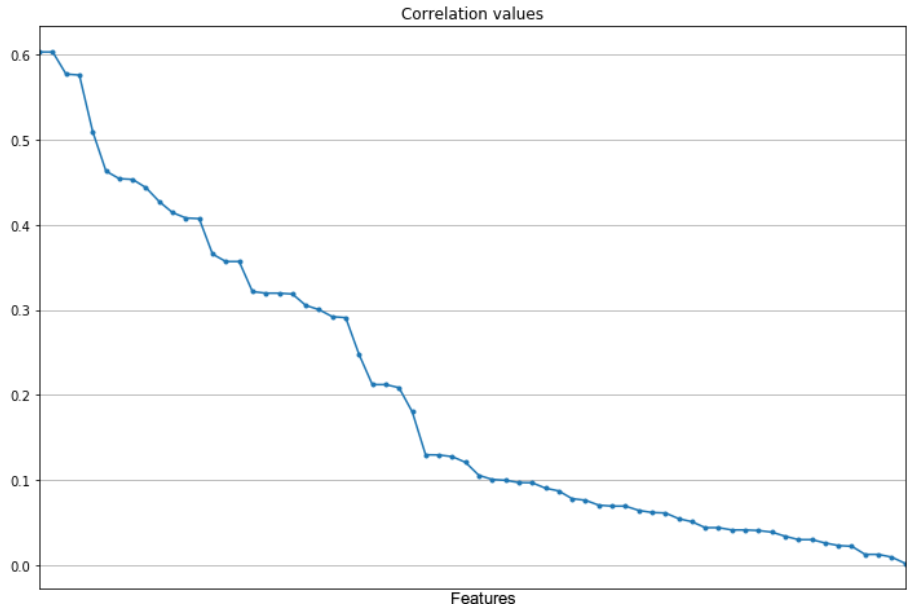

Figure 3.7: Correlation values for each feature in the dataset

As we can see, there is a small set of features whose values have a correlation higher than 0.5 with the value target value. A gap can be seen between these set of features and the rest of the features that compose the dataset. In order to improve scalability and avoid dimensionality issues, we are going to select these features in order to learn machine learning models.

These features are:

- Bwd Packet Length Mean: Mean length of packets in a flow coming in backward direction
- Avg Bwd Segment Size: Average size observed in flows coming in backward direction
- Bwd Packet Length Max: Maximum size of packets coming in backward direction
- Bwd Packet Length Std: Standard deviation of the length of packets coming in backward direction
- Destination Port

We can take a closer look at some of these features in order to understand the values obtained in the correlation analysis. For example, if we analyse the histogram that represents the values of the "Bwd Packet Length Mean" feature for entries labelled as *Attack* and as *Benign*, we can observe that, while most of the *Benign* entries are valued lower than 1,000, a significant amount of entries associated with attacks have values that range between 1,000 and 4,000. This can be seen in Figure 3.8. The "Avg Bwd Segment Size" feature presents a very similar distribution.

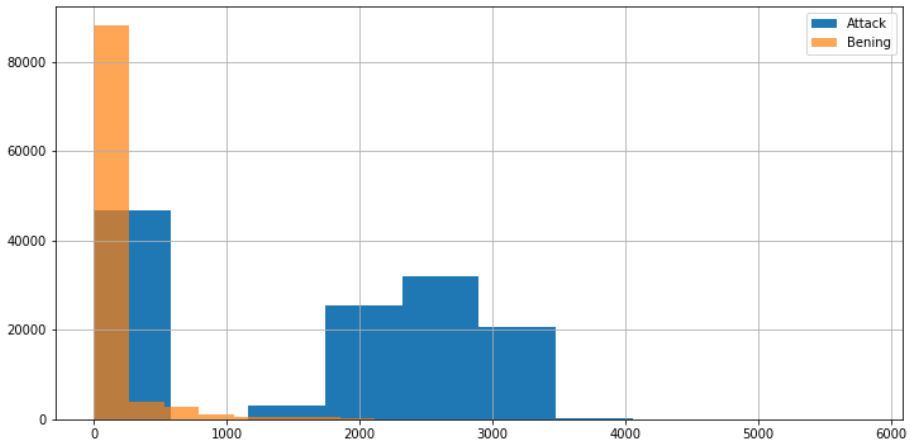

Figure 3.8: Histogram portraying the mean length of packets in backward direction

Regarding the feature "Bwd Packet Length Max", almost all of the entries that show *Benign* traffic flows have a value of 2,000 or less. However, the distribution of values regarding flows labelled as *Attack* is quite more spread. While most of attack flow entries also have values between 0 and 2,000, there is a significant amount of entries with a value higher than 3,000, as we can see in Figure 3.9.

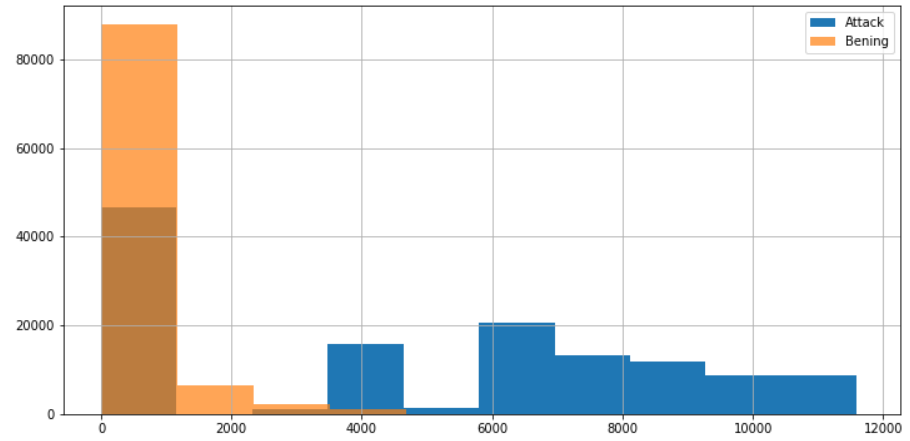

Figure 3.9: Histogram portraying the maximum length of packets in backward direction

A similar statement could be made about the standard deviation of the length of backward packets in flows. Traffic flows associated with benign traffic are very consistent regarding their packet length; therefore, most of the packets associated with benign flows maintain a very similar packet length, and their standard deviation is small in comparison with attack-labelled packets. On the other hand, flows associated with attacking actions can be divided between a first set of flows with packet lengths following a small standard deviation, and a second set with standard deviations higher than 2,000. These distributions can be seen in Figure 3.10.

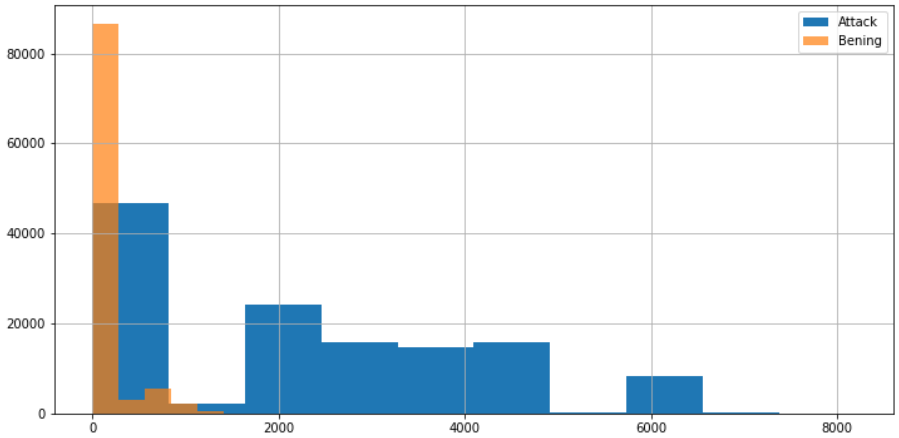

Figure 3.10: Histogram portraying the standard deviation of the packet length in flows in backward direction

Finally, regarding the "Destination Port" field, the difficulty resides in the fact that, while the attacks were performed against a small set of ports, most of the benign traffic also is directed against those ports. Therefore, it is difficult to distinguish between attack and benign traffic using only this field.

Once we have described the most important aspects of the variables used to learn the Machine Learning models, we test multiple models and analyse the results obtained. In order to test those models, we extract 25% of the data available for testing. We will use the F1-Score parameter (which is a weighted average of the precision and the recall of the algorithm in the classification process) in order to select the best algorithm. A table comparing multiple Machine Learning algorithms can be seen at Table 3.1.

As we can see, most of the high-performance classifiers are ensemble-based methods. This methods are based on the use of multiple machine learning algorithms that make predictions (independently or not, depending on the ensemble technique) using most or all the data available, and then the predictions made by each model are aggregated.

Algorithm	**F1-Score**
Random Forest	0.985071
Extra Trees Classifier	0.985070
SVC	0.985054
AdaBoost	0.985010
MLP Classifier	0.983062
Decision Tree Regressor	0.967289
Logistic Regression	0.759716
Gaussian Naive Bayes	0.749712

Table 3.1: F1 score obtained after using multiple algorithms for the prediction task

As we can see, the best result has been obtained using specifically one of these ensemble methods known as **Random Forest**. This method consists in using multiple decision tree classifiers that extract (with replacement) subsets of data and performs prediction over them. Then, the set of predictions are averaged and the resulting average is the final prediction being made. Several parameters regarding the decision trees being used can be tuned; this tuning process will be performed next.

In order to perform this process, we define a range for the parameters that can be configured in the Random Forest Classifier. First, we are going to define two possibilities for the split criterion of each tree when it branches out. These two possibilities consist in using either the Gini impurity or the Entropy. The Gini score values the split of each branch depending on how mixed are the classes resulting of such split. On the other hand, the Entropy criteria values the split of each branch using the *Information Gain*. The Information Gain of a split is defined by the difference between the entropy of the original branch and the average of the ehntropies of the resulting leaves.

We are also going to test multiple values for the min_samples_split parameter. This parameter defines a minimum number of samples required to split a node. Since the dataset used has a high quantity of entries, we predict that small variations in this number, specially when this number is small (lower than 10), will only have a significant effect on the final leaves of each Decision Tree. However, large variations might have a significant effect over the whole tree.

We will test multiple values for the max_features field too. This parameter controls the number of features to consider when looking for the best split at each branch. Since we have selected only five features to build the Machine Learning model, we will test values from one to five to test the effect of changing the number of features considered in the F1-Score.

Finally, regarding the Random Forest classifier overall, we can choose if we use the whole dataset to build each tree or we just bootstrap samples from the original dataset for each of the trees that compose the Random Forest classifier.

The results of this process can be seen in Figure 3.11. Each point represents a different combination of the parameters stated in the previous paragraphs. Even though the graph could lead us to think that there is a wide variation of scores among combinations, we have to keep in mind that the Y axis is ranged between 0.9808 and 0.9816. Therefore, there is little variation of the F1-Score among combinations.

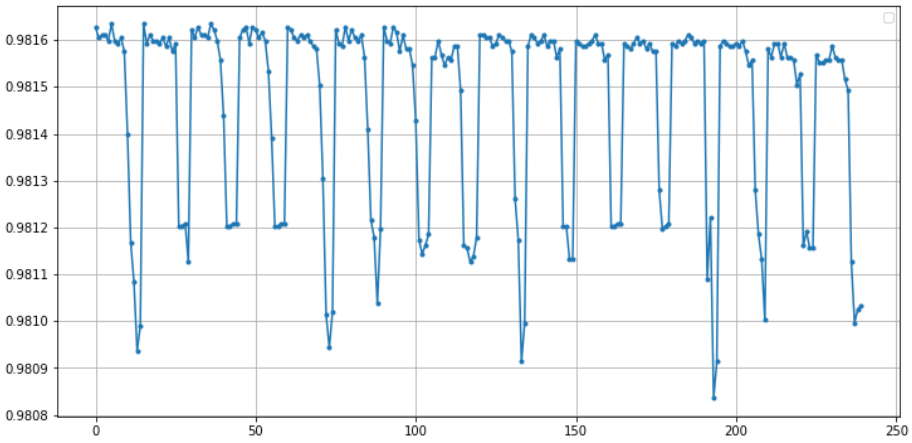

Figure 3.11: F1-Score obtained for each combination of parameters tested

The highest scoring combination of parameters obtained a F1-Score of 0.981634 over the training dataset, with a value of 7 for the min_samples_split parameter, 1 for the max_features parameter, Gini as the splitting criteria and bootstrapping activated. Once we test this model over the testing dataset (the one used for obtaining the scores showed in Table 3.1, we obtain a F1-Score of 0.985102.

As we explained during the description of the Random Forest Classifier algorithm, it is based on the combination of the results of multiple Decision Trees. One of these trees can be seen at Figure 3.12

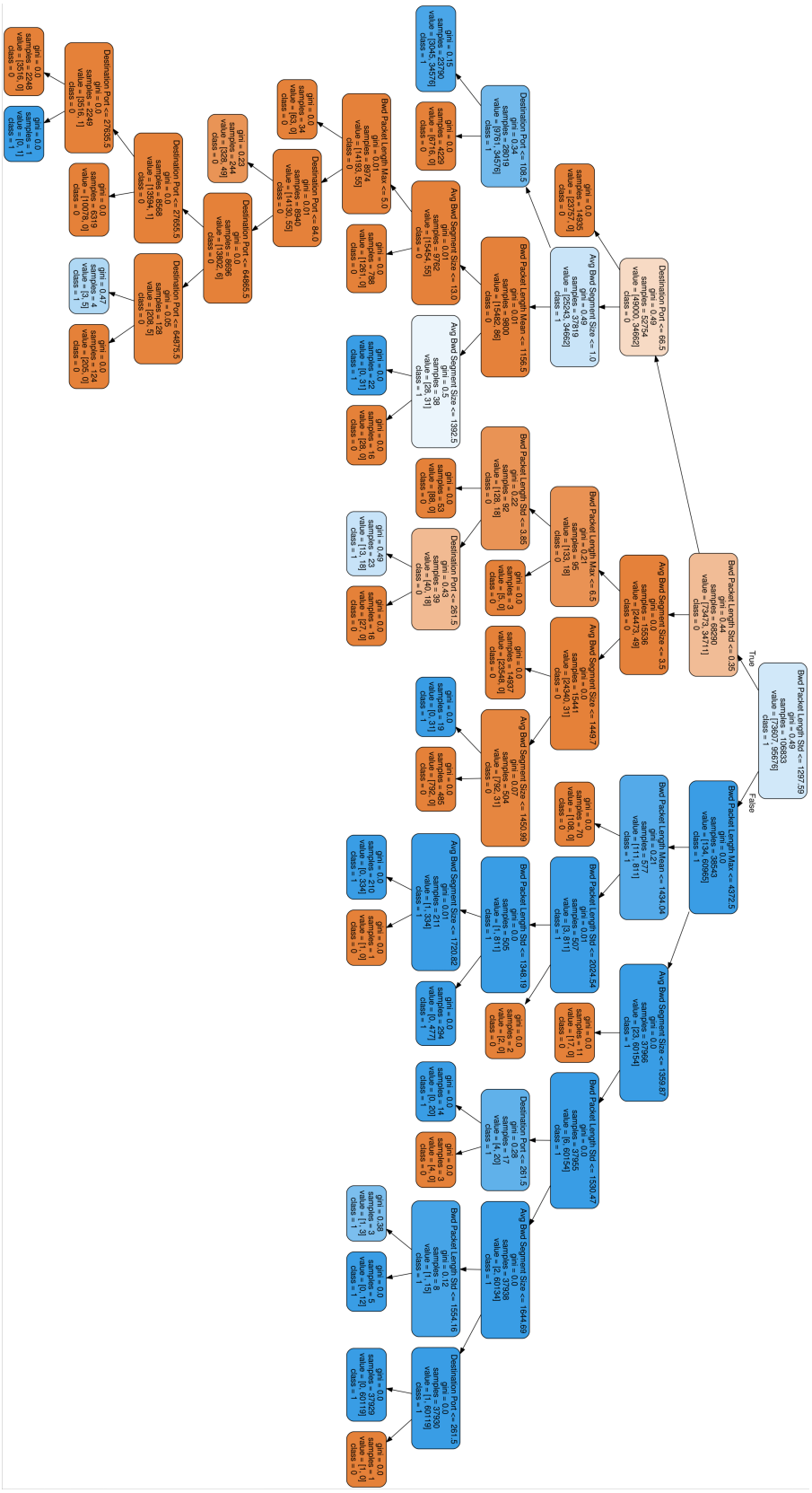

Figure 3.12: Decision Tree used in the Random Forest Classifier

CHAPTER 4

Architecture

In this Mater Thesis we have designed and developed a system for the detection of attacks within an SDN environment and designing of a suitable and scalable response to such attack. In order to perform these tasks, we need to lay out an architecture with the appropriate set of tools that allow us to perform such tasks. This involves adapting many tools related with Linked Data and Big Data technologies to SDN tools, including data pipelining and a visualisation system.

This chapter describes the architecture defined for this project. First, we describe the modules that will compose the proposed architecture, putting an emphasis in the task implemented by each subsystem. Then, we describe the implementation and connection of these subsystems, obtaining the complete architecture of the system described in this Thesis as a result.

4.1 Introduction

The general architecture of the system developed in this master thesis is based on the concept of *Data Lake*. A Data lake is a centralised system in the general architecture where data obtained from multiple sources following different formats is injected. This data lake stores vast amounts of data with different formats and offers multiple services related to data storing, such as indexing, querying support and integration with other data tools.

This chapter is divided in the following sections: first, in Section 4.2, we describe the general architecture of the system. In Section 4.3, we describe the architecture of the data ingestion layer tasked with the ingestion of data into the Data Lake. Section 4.4 describes the subsystem that includes all tools involved in the initial processing of data. Section 4.5 presents the subsystem tasked with the semantization of the data collected according to the semantic models already reviewed. Next, Section 4.6 describes the inner workings and fitting of the attack detection module in the general architecture. Finally, Section 4.7 portrays the inner workings of the dashboard developed for this project.

4.2 Architecture overview

The general architecture of this project is based on a modular conception of the tasks to be implemented. Therefore, the main tasks have been roughly equally divided among different modules. The resulting architecture designed with this criteria on mind can be seen in Figure 4.1.

As we can see, the main system of the architecture is the Data Lake, which interacts directly or indirectly with the rest of the modules. Therefore, this module is placed in the center of the architecture. Data is fed to this module by the data ingestion layer. This layer takes data directly from the network through the SDN controller and store it in the Data Lake module. Depending on the type of data, it is ingested following different predefined pipelines. Some of this data is processed and appended to the raw data being injected into the Data Lake. Therefore, if we want to obtain a global view of the current status of the network we only need to query one module: the Data Lake.

This statement can be observed in the fact that every other service implemented in the architecture (the semantization service, the attack detection service and the visualisation service) require a single link to the Data Lake module to collect the required data.

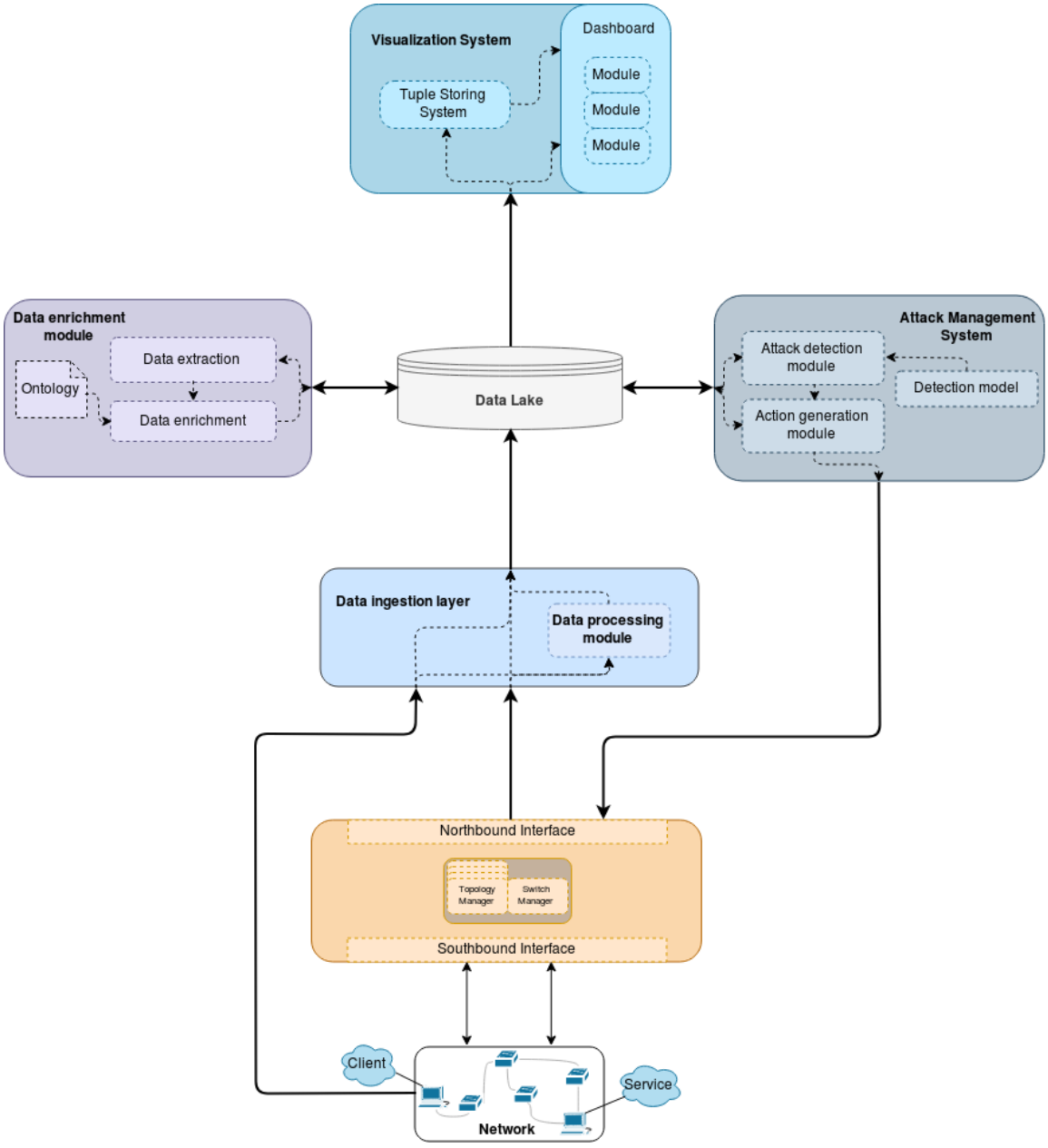

Figure 4.1: Architecture Overview

4.3 Data Ingestion layer

The data ingestion layer can be implemented following two principles. The first method would consist in implementing our own querying system, surpassing or complementing the data provided by the SDN controller. We could query directly network elements, such as hosts or nodes, in order to obtain data that the SDN controller was not programmed to monitor. This approach has the advantage of potentially accessing data that would be

otherwise inaccessible. This approach could also ease a transition from SDN environments to legacy networks. However, this approach enters in conflict with the principle of centralising the network management in one point, since we are overriding the network controller and accessing each element directly.

This approach could also entail scalability-related issues, such an exponential increase of network and computing resources requirements parallel to the increase of network elements, since the data ingestion layer would have to control the monitoring of a ever-larger number of nodes.

The second approach consists in using data already collected by the network controller. Using queries targeted at its northbound interface, we can access data provided by services already implemented in most SDN controllers, such as host trackers or interface monitors. This data can contain enough information for us to infer the presence of a current attack within the network, while also maintaining the scalability features provided by the centralisation of network management tasks in the network controller. Hence, we would be successfully protecting the complexity of the data ingestion layer from the potentially limitless increase of complexity in the network being managed.

As we can see, each approach has its own advantages and drawbacks, which makes it difficult to choose a method without knowing specific details regarding the environment to be monitored. However, we can follow a "hybrid" approach, where we also collect data directly from network elements instead of just querying the network controller, but we only query a limited number of elements independently of the size and scale of the network. Collecting data from only a small set of elements limits considerably the scalability issues associated with the first method. This method will be followed when implementing the data ingestion layer.

Specifically, we will monitor a small set of interfaces directly connected to hosts that provide the services that we want to protect within the network. We will also query periodically the northbound interface of the SDN controller in order to obtain information regarding the general status of the network. This data will be directly injected into the Data Lake, while some of this data will be also processed with the goal of using it to detect network attacks.

Hence, the monitoring process will be composed by two processes that represent the two sides of the hybrid monitoring approach. This process can be seen in Figure 4.2. As we can see, the ingestion process is divided in two flows associated to each of the sources listed before. The first flow collects data directly from elements within the network, while the second flow queries the data stored by the SDN controller regarding the current status

of the network.

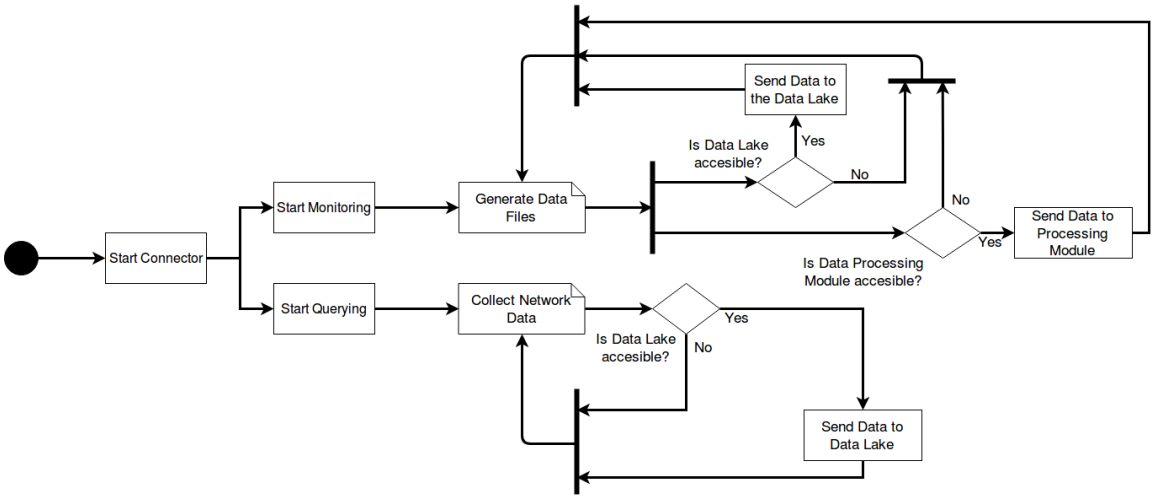

Figure 4.2: Activity diagram of the data ingestion layer

4.4 Data Processing Module

In this section we describe the data processing module that can be seen in the system architecture (Fig. 4.1). In the context of attack detection, processing tasks must be focused in the extraction of useful features regarding traffic flows being currently active within the interfaces being monitored. However, this entails the creation of a pre-processing step where raw data extracted from network interfaces and nodes is aggregated into flows. This data should also be included in the Data Lake, since it contains the features used in the development and deployment of the Machine Learning model tasked with the detection of attacks being performed in the network.

In the context of traffic flows associated to TCP connections, the development is relatively simple. The focus should be put in monitoring the presence of TCP control sequences. By monitoring the timeline of SYN and FIN messages within TCP connections (and their respective ACK messages) while grouping them by origin and destination addresses and ports, we can obtain aggregate single packets into TCP flows in an accurate manner.

However, when processing UDP flows, the need arises to establish a timeout per origin-destination pair, since the UDP protocol does not have a mechanism to establish connections like TCP does. The same goes for any protocol with a connection-less scheme.

As a consequence of the statements made in the previous paragraphs, the processing of data regarding packets must be done in batches, since using aggregated packets during a

pre-determined period of time is the only way of relating TCP connection control messages or detecting timeouts over a set of related UDP packets. This entails a small delay in the detection of flows in real time, since we need a set of accumulated packets before starting the processing tasks. However, we can configure this "delay time" depending on our intentions: we can reduce this time in order to improve the real time capabilities of our system, or we could increase it in order to detect traffic flows with a long duration in time.

4.5 Data Enrichment System

In the previous sections, we described the process and systems involved in the collecting of raw data needed for detecting network attacks. Furthermore, we described the precise flow of data in Figure 4.2. However, most of this data is raw; therefore, it lacks flexibility regarding possible future uses of such data. We could increase its flexibility by designing and applying ontologies in order to enrich such data by semantizing it and thus provide a structure for it. In this section we will describe the Semantization Module, which is tasked with such duties.

As we have previously explained, we made use of multiple ontologies regarding SDN scenarios and cyberattaccks. Therefore, the architecture of such process involves processing and semantizing multiple batches of data following different flows.

Specifically, two different data flows have been set up regarding the data extracted and processed in previous step. The first data flow is related to the data extracted regarding the general status of the network. As we have previously stated, this data allows us to obtain a wide view of the current status of the network. Depending on the use case, we might want to focus on specific areas of the network, such as traffic statistics or services monitoring.

The proposed architecture supports the possibility of shifting the specific areas to which this "first flow of data" references. In order to control the significance of such data, changes must be applied to the data ingestion layer; specifically, changes regarding the queries being sent to the SDN controller. Obviously, the ontology being applied must be adapted to the data being processed.

The second data flow represents the semantization process being applied to data regarding attacks being monitored or detected in the network. This data is also collected from the Data Lake; however, it is injected into the Data Lake by the Attack Detection System (which will be described in Section 4.6).

Semantizing data regarding attacks being detected in the network is useful not only for the reasons previously listed (flexibility and standardisation of data), but also because it

will facilitate the listing and querying over attacks being detected in the network. This is a fundamental feature in any attack diagnosis system, since obtaining a global view of the attacks being performed in the network while also being able of designing complex queries in order to select only a small subset of attacks.

In Figure 4.3, we can observe a diagram representing the flow dividing mentioned in the previous paragraphs. As we can see, each type of data is semantized in a different data flow, depending on the subject of the data. However, the semantized data is later stored in the data lake, independently of the data path followed. Lastly, the process is repeated by querying again the Data Lake.

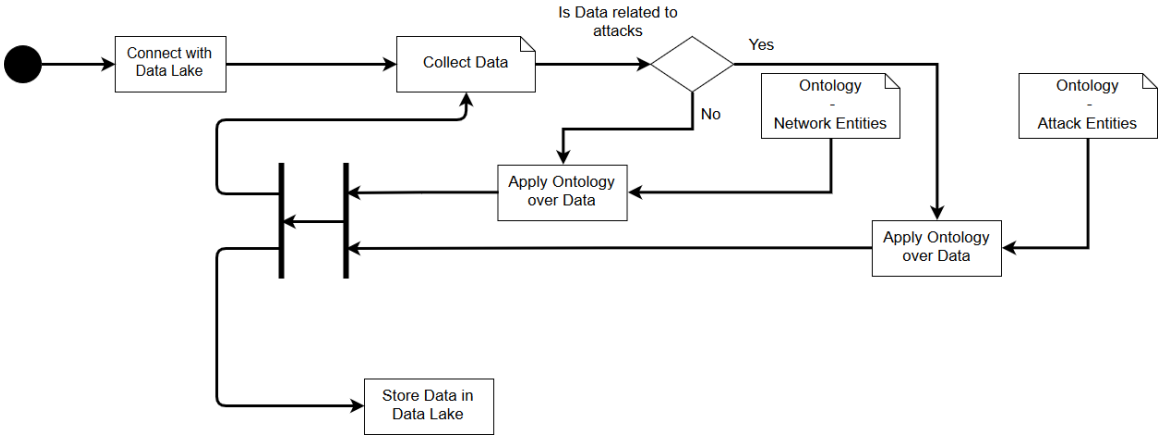

Figure 4.3: Diagram representing the semantization process

Regarding the semantization process itself, it basically consists in extracting certain fields from the data collected from the Data Lake, in order to assign values to each entity, property or relationship that is being specified in the ontology used for such purpose in the architecture. Therefore, it involves going through potentially vasts quantities of data fields in order to select the ones that suit your ontology.

In fact, there is a trade-off between the reach of the ontology models being used in this architecture (and thus, the quality and scalability of the structured data after it has been semantized) and the size and scale of the data being semantized. Therefore, the objective should consists in finding an equilibrium between the size of the data and the reach and size of the ontologies being applied to the semantization process.

However, we can also use the indexing features of the tools and systems used in the Data Lake in order to build complex queries that allow us to select only the specific data needed to instantiate the ontologies being used. By building such queries, we can avoid scalability issues regarding the amount of data being processed while also applying semantic models that represent extensively the network being monitored.

4.6 Cyber-attack Detection Module

The attack detection module is one of the most important modules present in the proposed architecture. It holds most of the " intelligence" present in this architecture, since it takes on the task of monitoring each batch of data being pushed into the Data Lake by the Data Ingestion Layer for the purpose of executing the reasoning process using the Machine Learning model trained for such task. Therefore, it maintains connections with both the Data Lake and the SDN controller.

This module not only addresses the task of detecting attacks being performed in the network; it also communicates directly with the SDN controller in order to perform mitigating actions that limit the effect that the attacks being performed in the network might have over the services being provided in it.

Instead of using a module that controls and coordinates the communications between each module of the system proposed and the SDN controller, the decision has been taken to allow the attack detection module to communicate directly with the SDN controller for the purpose of allowing a much faster reaction to the attack being performed. In this scenario, as soon as the Machine Learning model predicts the presence of an attack associated to a data entry collected from the Data Lake, it will perform the necessary actions to block such attack while also providing the services currently being provided in the network.

These actions, which are partially pre-programmed into the attack detection module's code, could range from trying to block all the traffic coming from the direction associated to the attack, to trying to accommodate the attack by re-routing it to other network elements for the purpose of giving the attacker the illusion that his/her attack is being carried successfully while it is not affecting the provisioning of the services.

In a legacy network environment, this criteria could translate into developing a complex software system tasked with calculating alternative blocking or re-routing rules and distributing these rules among each network node separately, thus making the whole reactive process significantly slower. Furthermore, such complexity could also carry significant scalability issues, since the resources needed for such process would increase exponentially with time.

However, thanks to the SDN paradigm, the attack detection module only needs to build a set of high-level rules in order to either block or re-route such attack and send them to a centralised point. The task of coordinating the updating of each node in the network is being faced by the SDN controller in this architecture.

As we can see in Figure 4.4, the process is based on the existence of a Machine Learning

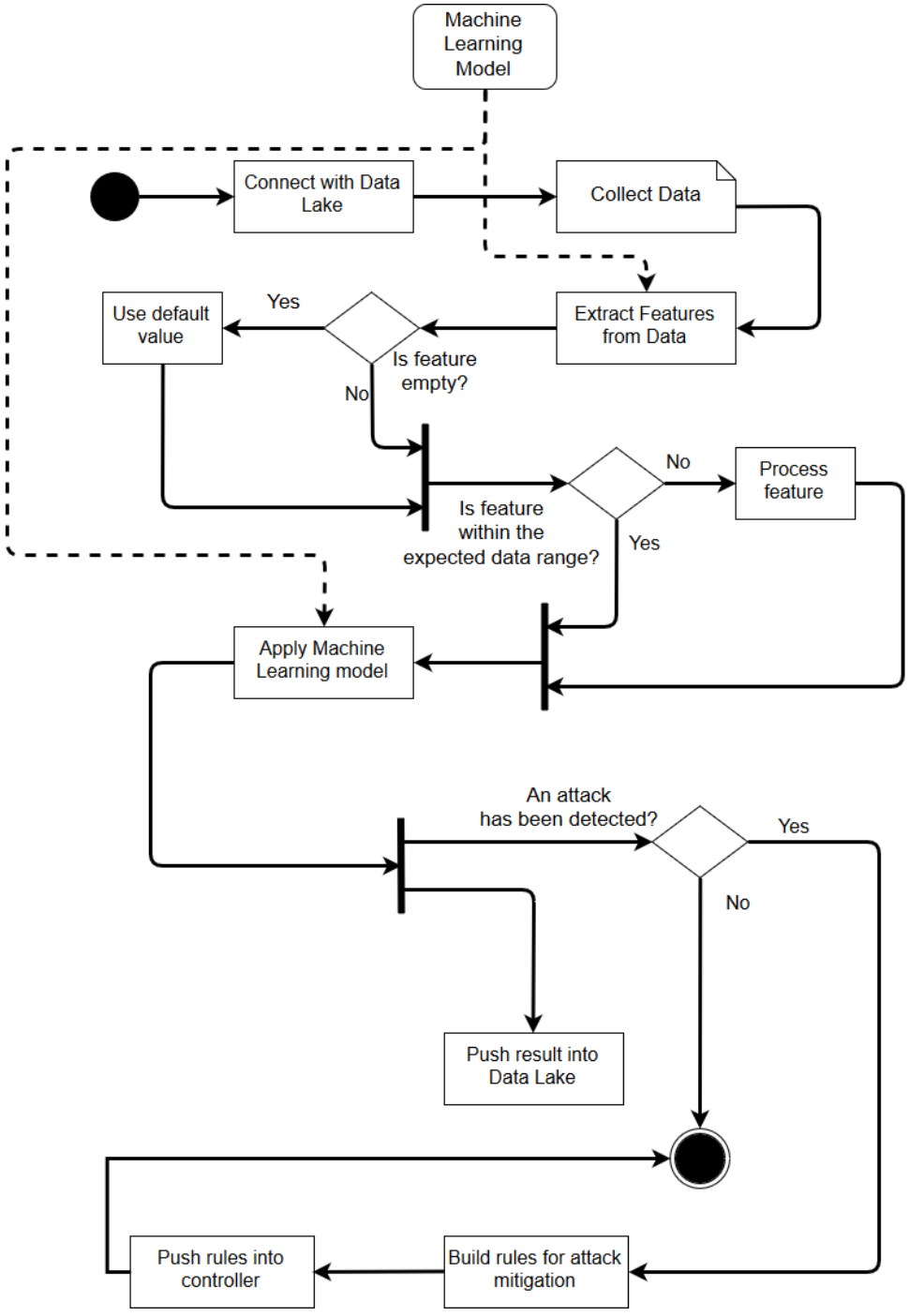

Figure 4.4: Diagram of the attack detection procedure

model which requires a set of features in order to make a prediction regarding the presence of attacks.

4.7 Visualization Module

Using the modules described in the previous sections we already can apply the core functionality of monitoring an SDN network to the detection and mitigation of cyberattacks, while also providing a semantic framework for the data collected regarding both the current status of the network and the attacks being detected.

However, current methods for accessing and monitoring such data are very stiff. If we want to access data that shows statistics on the current use of the network or a listing of the attacks being detected, we have to compose complex queries in order to select exactly the data required. We also have to get in contact with the Data Lake in order to send such queries; most of the times this means using a REST API which has not been designed with the goal of providing a satisfactory user experience.

Therefore, we consider that a visualization module is needed in order to greatly improve the usability of the proposed system. Regarding this, one of the possibilities is consider the use of the Elastic Stack mentioned in Section 2.3.1. This can be considered as an appropriate solution, since ElasticSearch can be considered for the role of Data Lake (and in fact it will, as we shall see in Chapter 5) and therefore we can consider Kibana for the role of dashboard.

Kibana is a tool provided by the Elastic Stack which provides a Web interface for the purpose of monitoring data stored in ElasticSearch. Since it is specifically designed for working with ElasticSearch, it eases significantly the querting and observation of data stored in ElasticSearch. However, it is too broad for the purpose of this project: we would like to focus on specific fields regarding statistics on the current status of the network and the presence of attacks.

Hence, another option consists in the development of our own visualization module. For this purpose, we have selected and further developed a dashboard based on the Sefarad architecture mentioned in Section 2.5.2. The Sefarad visualization system allows us to represent data collected directly from the ElasticSearch module while also controlling the default queries and data collected to be presented.

This represents an improvement over the default Kibana visualization system (which shows all the information present at the ElasticSearch module, circumstance that can be overwhelming to the user of this system). However, we would also like to use a semantic data indexing system that would allow us to use SPARQL queries. This allows us to use a pure semantic approach to the treatment and visualization of semantic data, further enriching the flexibility of the visualization system.

In order to make use of these SPARQL queries, we have selected the Apache Jena Fuseki technology already mentioned in Section 2.4.2. In Jena Fuseki we will store the triples representing the semantized data in RDF format, for the purpose of allowing the execution of SPARQL queries by the visualization system. This system will execute queries in order to collect specific fields regarding attacks and the network status, instead of collecting widespread data. The architecture of the resulting system can be seen in Figure 4.5.

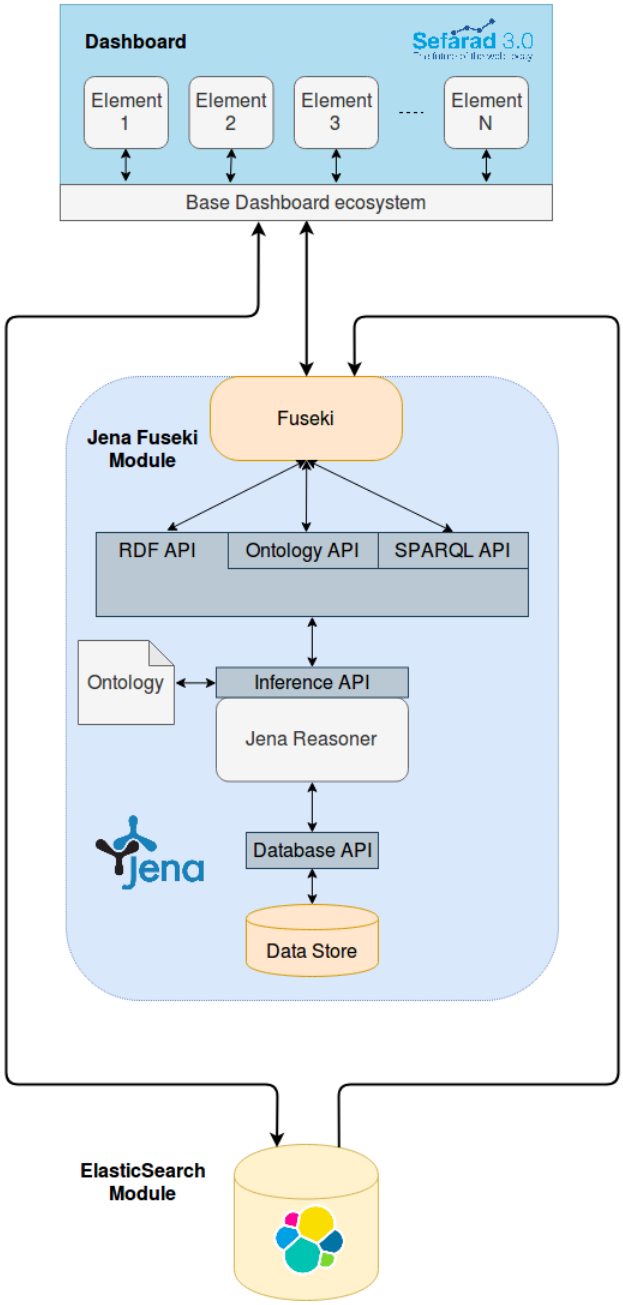

Figure 4.5: Visualization Module Architecture

As we can see in Figure 4.5 all the interactions performed with the Jena Fuseki module are carried out through Fuseki, which controls and redirects all the HTTP connections sent to the Jena Fuseki module. First, the semantic data stored in the ElasticSearch module by the Semantization System mentioned in Section 4.5 is collected and stored in the Tuple Storing System also known as Jena Fuseki Module. This is done by creating a database and sending each entry representing a semantic tuple. Then, the Jena reasoning core checks the validity of such tuples and stores them in the in-memory data store. Finally, when the Dashboard composes and sends a query in order to collect specific data, the reasoner core analyses the query received via Fuseki (since this query is sent using the HTTP protocol) and returns the queried data.

Then, the dashboard send queries to both the ElasticSearch module and the Jena Fuseki module in order to collect the specific data required. The querying of data is controlled by the Base Dashboard ecosystem. This submodule holds most of the logic required by the dashboard and coordinates the functioning of each subelement present in the dashboard.

The role of the base dashboard ecosystem consists in offering a framework where dashboard elements such as charts, graphs and lists can be easily added or removed. Furthermore, this framework provides these elements with the data needed for the representation of the information associated to each element. It also coordinates the execution of each element when new data is needed.

This process can be seen in Figure 4.6.

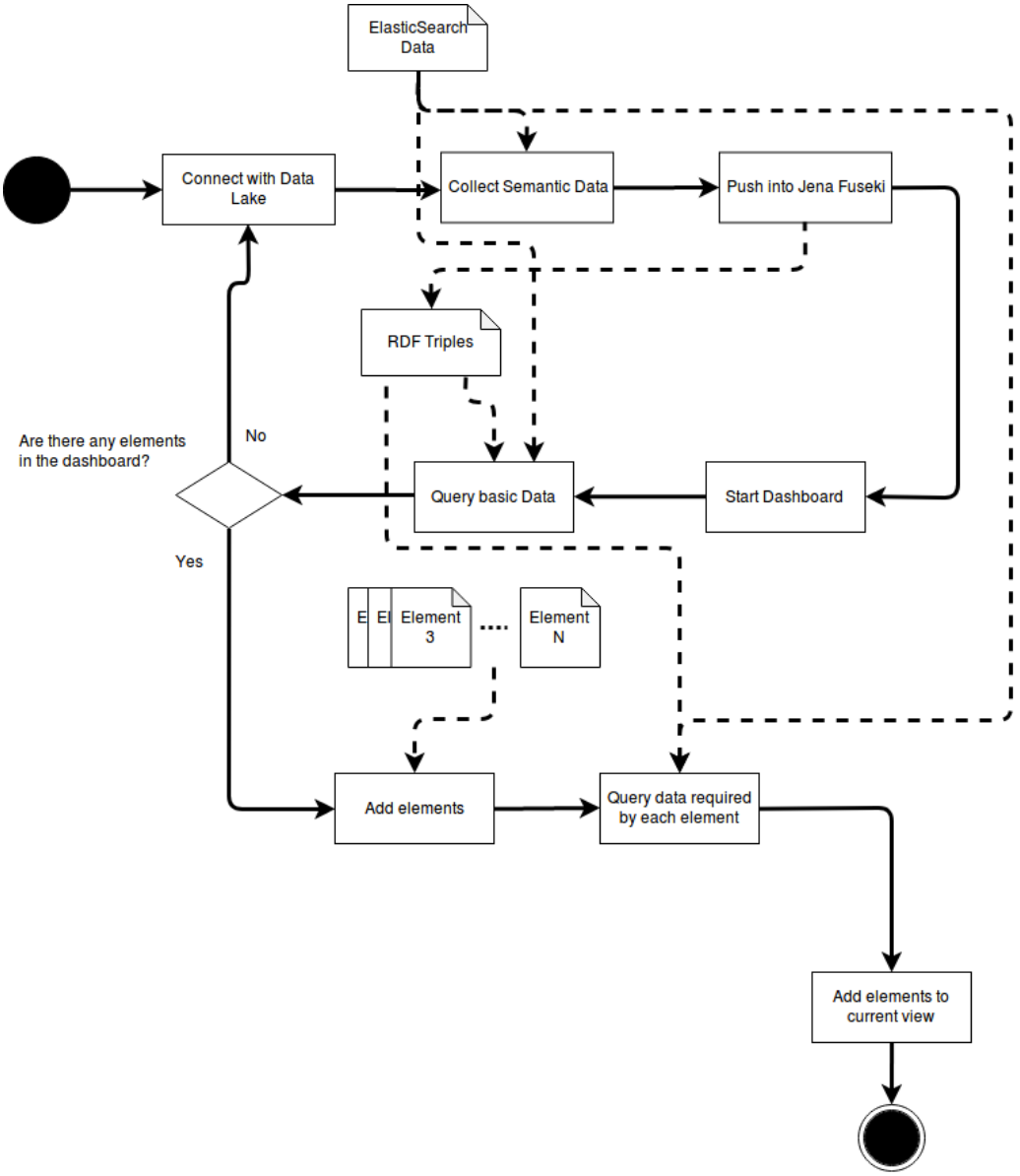

Figure 4.6: Diagram showing the collection and visualization of data

CHAPTER 5

Case study

As we have previously stated, the SDN paradigm allows us to notoriously increase the flexibility and quickness of current network managing techniques. This includes the collecting and monitoring of information regarding possible attacks within this network, as well as measures taken when attacks are detected in order to either block or re-route such attacks. These advantages are heavily responsible for the current increase in the adoption of SDN technologies.

Therefore, in this chapter we are going to describe an use case related to the goal of detecting and blocking attacks in such environment, as well as semantizing the resulting data and visualizing the results. The system developed for this use case will be based on the architecture presented in Chapter 4.

In this chapter we will get into details regarding the specific network being virtualized in order to be used as a testbed. We will also explain the deployment and execution of the SDN controller, as well as the modules responsible for the data processing and pipelining. We will also detail the use of the processed data in order to execute the Machine Learning model for the purpose of detecting attacks. Finally, we will describe the semantization implementation of such data, as well as the visualization module developed for such purpose.

5.1 Introduction

In this and the following sections, we will describe the use case in the thesis in order to build a prototype. The architecture described in Chapter 4 has been used as the basis for the development of the prototype. This prototype creates a virtualized SDN scenario controlled by Opendaylight which will serve as the testbed for the rest of the system.

Then, a data ingestion layer is implemented. In this layer data is collected from multiple sources within the network. The SDN controller also represents another important data source, specially regarding the current overall status of the network. While data from those sources is being collected, some of this data is also being processed in order to be used by the attack diagnosis system. Therefore, two different pipelines are laid out for each flow of data being collected from the testbed.

Then, both raw and processed data are stored into the data lake module, which is implemented using Elasticsearch. This data lake will serve as the central data store for all the raw and processed data collected from the rest of the modules present in the prototype. An image of the implementation of such architecture can be seen in Figure 5.1. Once the data collected from the testbed is stored in the data lake, the attack diagnosis module collects the necessary data from the Elasticsearch unit in order to detect the presence of an attack in the network. This module will yield a result regarding the presence of an attack in the network; furthermore, it will also compose networking rules in order to block the attacks being detected. Then, both the data collected from the network and the data regarding the attacks are semantized following the extended semantic model described in 3.2.3. Finally, the semantized data is stored in Elasticsearch and the tuples are also fed to the Jena Fuseki tuple indexer. Finally, the visualization system collects data from both Jena-Fuseki and Elasticsearch in order to show an overview of the status of the network.

This chapter is divided in the following sections. In Section 5.2, we will describe the implementation and deployment of the environment where the test network is deployed, as well as the target service. It also describes the deployment of the SDN controller and its interactions with the network. Then, in Section 5.3, we will depict the inner workings of the ingestion layer and the different paths that a data flow can follow depending on its source. We will also describe the processing of the attack-related data for the extraction of features necessary for the diagnosis of attacks.

Next, in Section 5.4, we will describe the implementation of the attack detection module and the actions that this module takes in order to block the detected attacks. Then, in Section 5.5, we will describe the functioning of the scripts tasked with the semantization

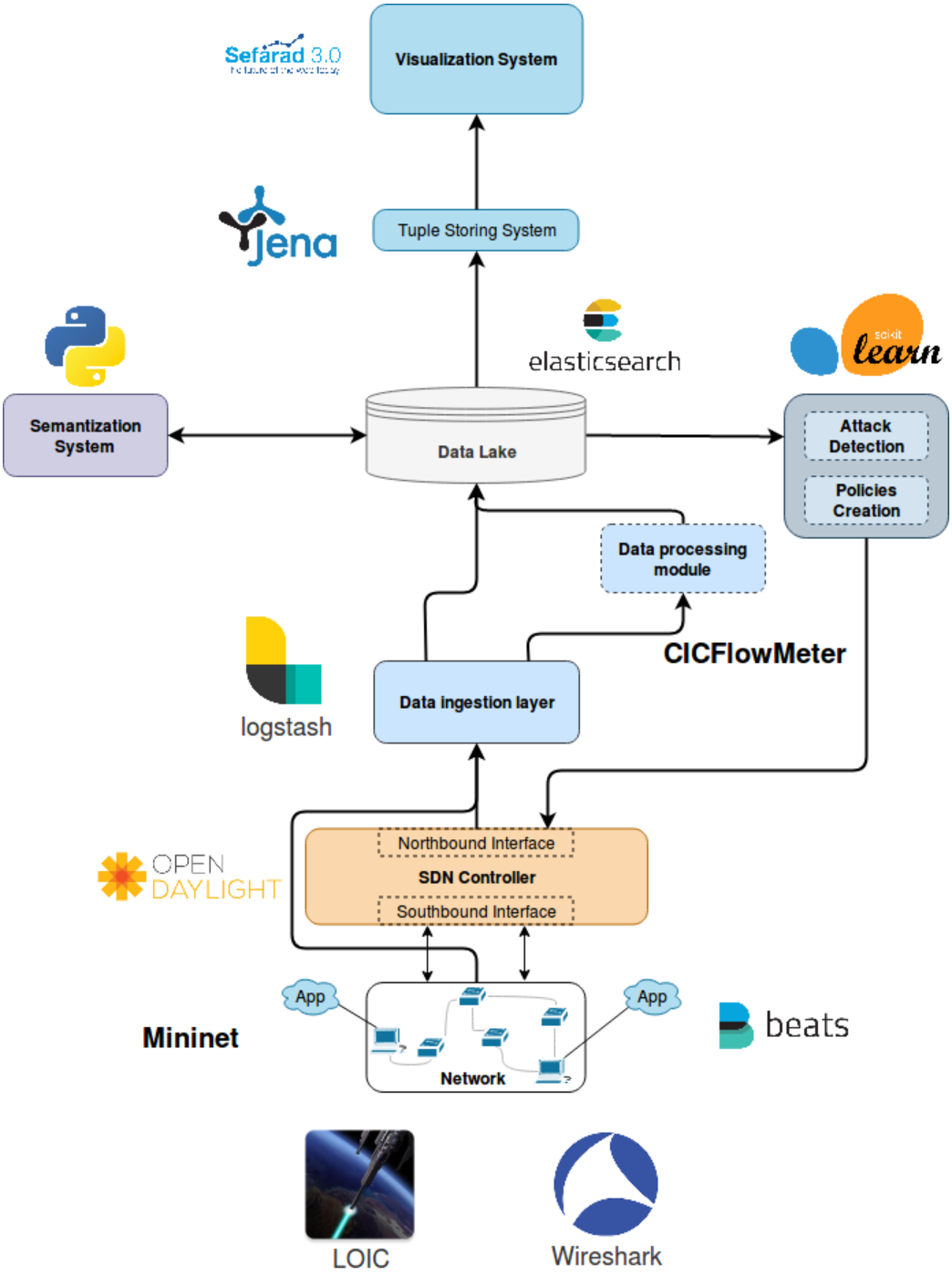

Figure 5.1: Prototype architecture

of data. Finally, in Section 5.6, we will describe the indexing of tuples created by the semantizing module and the visualization of data implemented by the dashboard.

5.2 Network Environment

In this section, we are going to describe the network and controller modules deployed in order to provide a testbed to obtain data and an environment to test the developed systems. This description will include the detailing of the network deployment and the attack being implemented in such network

This section is divided in two subsections. In Subsection 5.2.1, we will depict the inner workings of the network being deployed. In Subsection 5.2.2, we will describe the deployment of the OpenDaylight controller and its interactions with the network being managed.

5.2.1 Software Defined Network

The goal of this project consists in learning a Machine Learning model that is able of predicting attacks using a labelled dataset provided by the Canadian Institute of Cybersecurity, and deploy such model in a manner that it effectively detects and blocks attacks in SDN. However, in order to test it, first we need to develop and deploy a SDN network where attacks can be performed and monitored. Therefore, we will focus first in the developing of this testbed.

First, we need a network composed by SDN-abled nodes. However, we do not have the necessary resources to deploy such network physically, so we have to virtualize it. For this purpose, we use a software tool known as Mininet, already described in Subsection 2.2.2. As we reviewed in such Subsection, Mininet allows us to virtualize an SDN while also specifying hyperparameters such as the topology of such networks and the protocols being used. Furthermore, we can also virtualize hosts connected to the network. This allows us to deploy services that use the network for connecting to clients represented by other hosts.

Specifically, we have deployed the network topology that can be seen in Figure 5.3. We have designed this topology with the purpose of selecting a host as the service provider and the rest of the hosts as clients and potential attackers. Therefore, by monitoring specific points in the topology, we can detect attacks and their influence in the providing of the service. By default, those hosts that do not provide a service act like clients; however, any client could turn into an attacker.

For testing purposes, we have developed a simple web server that returns a web page every time that it is queried. We have developed such system by programming a Python script. This server attends TCP connections, and it is programmed to keep them open

until it stops receiving messages through each connection separately connection. In that case, a timeout system is activated. If no new messages are received in a connection after seven seconds, the connection is closed, so the associated socket can be used to attend new connection requests.

As we have specified previously, we have used Mininet in order to deploy this network. Specifically, we have used the Mininet Python API. This API allows us to use the build-in *host* class to define clients, attackers and services within the network. We can either use the base *host* class (which virtualizes network hosts in a process-based manner) or extend such class in order to add parameters to it.

This API also allows us to specify nodes and links in order to build our topology. One of the features of this API that we used in order to build the topology consists in the ability of using links connected to traffic control interfaces. This enables the configuration of the network testbed in order to replicate behaviours present in real networks, such non-uniformity in links capacity within the same network, or irregular behaviours in different interfaces.

Regarding the nodes, Mininet uses Open vSwitch 2.2.4 running in kernel mode to switch packets among virtual Ethernet interfaces (created to emulate the interfaces and link defined in the Python scripts implementing the network topology). Since the network is deployed in a single computer (as we can see, Mininet virtualizes in such a manner that it does not allow a distributed deployment without the use of other communication tools), the switching of packets between hosts is very fast. Therefore, if we want to modify such behavior, we need to use traffic control interfaces.

Once we have deployed the network, we need to connect it to the SDN controller. This step will be further described in Section 5.2.2, along with the inner workings of said controller. However, once the tandem composed by the network and its controller is finished, we can test the functioning of such network and deploy software in the mentioned hosts.

Specifically, we begin by choosing a host and deploying a basic web server which was already introduced in Section 5.1. This server, implemented in Python, will attend HTTP GET and POST messages. It processes POST messages by reading and storing the information sent in such messages. On the other hand, it processes GET messages by replying with a simple Hypertext Markup Language (HTML) sample document.

Once we have deployed and executed this service in one of the hosts, it will listen to HTTP connections coming from the rest of the hosts. Then, we can access to these hosts by using the CLI provided by Mininet. Specifically, we can open xterm emulators for each of the hosts and send HTTP requests to the server to emulate real traffic. We have implemented

a five seconds timeout for TCP connections in order to allow an automated recovery of sockets which are occupied by "unfinished" TCP connections.

One of these hosts will be used as an attacker. As we mentioned in Subsection 2.7.2, we are going to use the tool known as Low Orbit Ion Cannon for performing the attack over the service provided. The Low Orbit Ion Cannon performs attacks by saturating web servers with a flow of TCP connections at a configurable rate. Hence, by configuring a rate that saturates the server and occupies all its sockets, we can block the services by not allowing it to reply to new connections from the clients.

However, the Low Orbit Ion Cannon is a tool programmed to be executed in a Windows-based environment. Therefore, we have two options: the first option would consist in using a virtualized Windows environment as host. However, this would entail deploying the Mininet-virtualized network in Windows, and any other tool used in the testbed would need to be Windows-compatible; hence, this option has been discarded.

Figure 5.2: Low Orbit Ion Cannon GUI

The second option consist in using a framework that allows the execution of Windows software. In this prototype we have used Mono [34], an open-source implementation of the Windows .NET framework that eases cross-platform development and deployment. Thanks to this tool we can access the Low Orbit Ion Cannon graphical user interface, which can be seen in Figure 5.2.

As we can see, we can configure many parameters regarding the DoS attacks generated by this tool. In this case, due to the limitations of the testbed environment, we will use the Manual Mode of the LOIC software, which allows us to turn a host into a DoS attack vector manually configured. However, in a physical environment, the LOIC tool gives us the

opportunity to create a HiveMind cluster where a single machine coordinates the execution of multiple LOIC instances in different machines in order to increase the optimisation of the saturating abilities of the DoS attack provoked by the attack system. The same goes for the Overloard configuration. They both allow the conversion from a centralised DoS attack to a Distributed DoS attack architecture.

Once the attack mode has been selected, we need to specify an IP address or an URL where the target server is located. Then, we can specify options related to the attack channels. For example, we can specify timeouts for the attack, in case that we only want to disable the providing of the service during a specified amount of time. We can also specify certain routes within the server being attacked in case that we want to direct the attack against an specific vulnerable location in the target server. We can also send messages along with the attack.

The last batch of options available in the Low Orbit Ion Cannon GUI also concerns attack parameters. Particularly, we can select an specific port target. The selected port by default is the standard HTTP port; however, we might want to attack systems where the HTTP server listens to many ports with the purpose of avoiding attacks that only consider web deployments using standardised parameters (for example the use of the port 80 as the port to which the server listens).

We can also choose among TCP or UDP in order to perform the attack. UDP-based DoS attacks have the advantage of not being built over an end-to-end mechanism. Therefore, it is harder for defending mechanisms to detect UDP flood flows when they are being sent form a massive set of source IP addresses.

However, TCP-based attacks have other advantages. Specifically, since TCP-based DoS attacks establish connections that last as long as the attacker keeps sending messages to the receiver. Therefore, while the goal of UDP attacks consist in saturating connections between the elements providing the service and the rest of the network, TCP-based attacks can also try to occupy all the sockets listening for TCP connections on the server side. Once each TCP connection is established, the attacker only needs to send TCP messages periodically in order to keep each connection open, without focusing on the rates of bytes per second being sent to the attack target.

Hence, while flows are easily detected by monitoring messages related to the establishing of TCP connections, systems based on the monitoring of byte rates find it harder to detect TCP-based DoS attacks. This approach will be followed in the implementation of the attack system in this use case.

Finally, we can select the number of threads dedicated to the performing of the attack

and the number of sockets per thread involved in such attack. These parameters control the output rates of the packets being sent by the Low Orbit Ion Cannon. If we use the TCP approach, we can also select the "Wait for reply" option, which makes the attacker wait for a response to each TCP message being sent.

In this use case, we are going to test the correct functioning of both the attacking and the detection tools by performing attacks from different locations, using a single host as attacker and using multiple hosts simultaneously. While we perform the attacks, we will also try to connect to the attacked service from other hosts in order to check the unavailability of the service being attacked and the successfulness of the attack being performed.

5.2.2 Opendaylight

As we described in Section 2.2, the SDN paradigm requires the presence of a network controller which has a centralized view of the network and controls the routing rules stored in each node. Mininet provides a default Python-based network controller known as POX [35]. This controller is a Python-based version of the NOX controller [36] written in C++. Even though this controller provides the basic functionalities needed for our network to route traffic successfully, we need a SDN controller that provides an interface which allows the access to the controller's database. Therefore, we have chosen OpenDaylight as the controller used in this use case. This SDN controller was already introduced in Subsection 2.2.5.

One of the advantages provided by the OpenDaylight controller is the fact that its architecture fits an abstraction layer between the modules tasked with controlling the communications with each element in the network and the rest of the controller. This allows for the representation of network elements as objects, which eases the configuration of such elements and the consulting of data regarding them. This layer is known as Service Abstraction Layer (SAL). Since network elements are represented as objects, modules and APIs can be easily developed in order to allow the provisioning network services installed into OpenDaylight and also provide an easy access to data regarding the network (each data element being modeled as an object).

One of the network services being used in this use case is the Host Tracker. This service monitors the number of hosts being present in the current simulation, and the status of each one. This is particularly useful since it allows us to build a system that can actively monitors the number of clients of our service. Furthermore, this information will be collected and used in the visualization system described in Section 5.6. These modules, along with the Switch Manager module (and Topology Manager module, which uses both the Switch

Manager and the Host Tracker) will provide the core of the data required by our system to provide the overview of the topology being monitored by our system.

Regarding the management of data within the OpenDaylight controller, OpenDaylight is configured in such a way that it distinguishes between two types of data states: operational data and config data. The operational data (defined by a concept known as Operational Data Tree) represents data regarding the current status of the network. Therefore, this type of data will be collected when querying the controller for information regarding the current status of the network. This type of data is updated by using reports directly collected from the SAL layer defined previously. Therefore, this data comes from reports being sent by the SAL either directly or indirectly (for example, when data is collected and processed by modules which use data from the SAL).

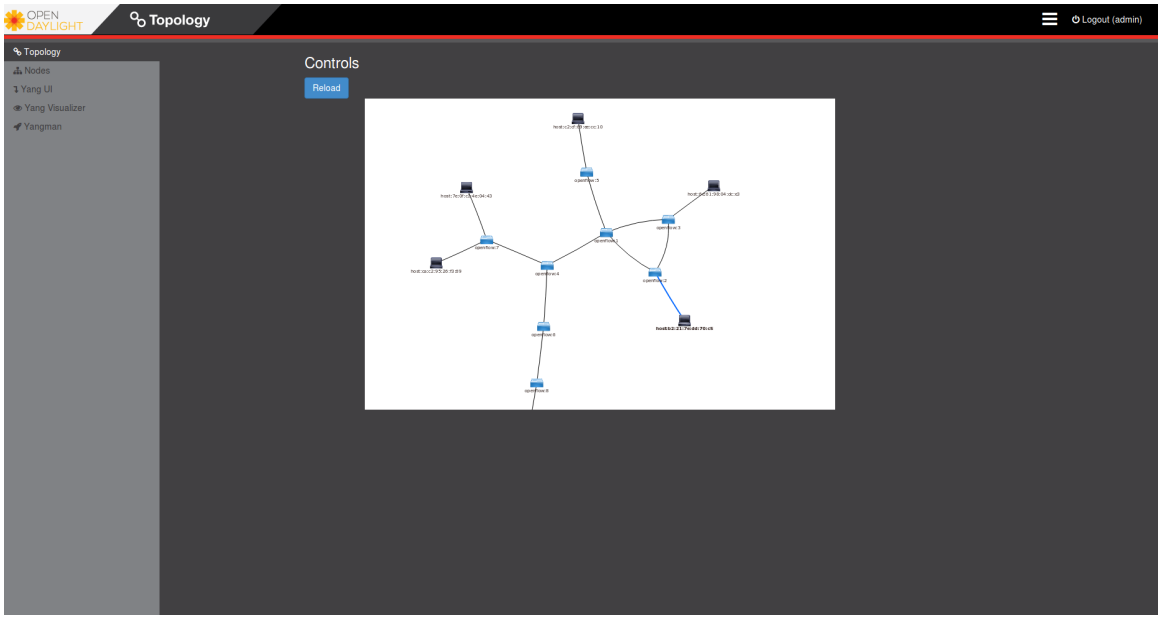

Figure 5.3: Opendaylight Topology UI

On the other hand, config data references data being pushed by the network administrator using either the REST API or an in-house module developed by such administrator. In this use case, we will push network traffic rules for the purpose of blocking ongoing attacks by querying the REST API in the northbound interface of the controller. Therefore, data entries tagged as config data represents data pushed by the attack detection module. This data can also be queried by other monitoring systems using the adequate queries.

Regarding the communications between the controller and the rest of the architecture, the southbound interface of the controller communicates with each network node through the use of the OpenFlow protocol already described in Section 2.2.3. Data queries are sent

to the northbound interface of the controller (specifically to its REST API) using HTTP messages over TCP. This method is also used to sent built routing rules to the controller, for them to be pushed into network elements specified in the messages being sent.

Once both the network and its controller are up and running, we can start the simulation of an attack and monitoring of the network by the system developed over it.

5.3 Data Ingestion

In this section we will describe the inner workings of the modules associated with the collecting and processing of data. We will also explain the tools involved in this process and how do they interact with each other.

This section is divided in two subsections. First, in Subsection 5.3.1, we will analyse the process followed in the collection of data form each source. Then, in Subsection 5.3.2 we will describe the processing steps followed for the data used in the attack detection steps.

5.3.1 Data Collection

In order to describe the tools and processes specified for the collection of data from the testbed, first we have to specify the sources involved in such data collection. In this use case we are specifically interested in data stored by the OpenDaylight controller, and data collected directly from interfaces connected to the service providers regarding packets being sent and received at such interfaces. Therefore, our goals will consist in developing collecting systems focused in querying those sources.

First, we focus on collecting data regarding packets flowing from and to the hosts representing service providers. In this case, we have several choices. For example, we could develop a module within the OpenDaylight architecture. This module could take advantage of the SAL layer provided by OpenDaylight and monitor the object "Interface" of an specific set of elements in order to register the data flowing across those interfaces.

However, this would force us to develop another module tasked with the decoding of data collected from those interfaces, since OpenDaylight would force us to work at a very low abstraction level. We would also be forced to work using Java, since most of the modules and interfaces already included in OpenDaylight are also implemented using Java.

Therefore, we have focused in collecting this information directly from the network, without consulting the SDN controller. Since we are using a virtualized network, we can easily access the container where the network is being virtualized and use Wireshark over

the interfaces connected to the service providers.

Wireshark [37] is a tool widely use for the purposes of monitoring traffic in computer networks and analysing flows in interfaces. It has the advantage of providing data at many levels of abstraction: it provides both byte streams representing data flowing through the interfaces and also the decoding of such data, therefore providing vital information such as the protocol of each packet, timestamps, sources, and other structured information contained in each packet.

By collecting data directly from elements within the network instead of through a centralized entity such as the SDN network, some might consider that an scalability issue might arise. However, since we do not monitor the entire network but only a small set of interfaces (those directly connected to the hosts acting as service providers), any scalability issue connected to the size of the network is heavily limited.

So, we configure Wireshark in order to monitor only the previously mentioned interfaces. While these interfaces are being monitored, Wireshark outpus the data into multiple files following two formats. It outputs the data in two different formats. The first format is Libpcap, which is the format associated to PCAP files. Files with this extension will follow the datapath which leads to the data processing module (which can be seen in Figure 5.1).

On the other hand, it also generates JSON files containing the same information. These files are generated to be pushed into ElasticSearch, since one of the main principles of a Data Lake based architecture consists in storing all the collected raw data in the Data Lake.

Once these two files are generated, they are monitored following different methods. Regarding the JSON files, they are collected by Beats, a tool that was already introduced in Subsection 2.3.3. This tool can be configured using a YAML file in order to select a set of files to be monitored. Beats is configured to monitor any new entries in those files, and it sends those files to the Logstash instance.

Logstash is a tool that was also described in Subsection 2.3.3. This tool receives data from multiple sources (such as beat messages sent by Beats) and processes them in multiple ways defined in a configuration file also written in YAML format. Specifically, we have configured Logstash in order to analyse if the data received comes from the JSON file or the PCAP file. When the data represents information stored in the JSON file generated by Wireshark, it is directly sent to the Data Lake implemented with Elasticsearch for its storage.

So far, we have described the data collecting process for data queried for attack detection purposes. However, we would also like to collect data about the general status of the network

in order to provide a context in our visualization system. Therefore, we will also describe the process of collecting data from the network controller.

As we have previously mentioned, the OpenDaylight controller provides data through the REST API present in the controller's northbound interface. In order to ease the access to the data provided by the API, this data has been modelled using YANG. Hence, by consulting the YANG models used by OpenDaylight beforehand, we can know the structure that the data queried to OpenDaylight is going to follow. This allows us to further "tune" queries regarding specific data to be used in te use case.

We are interested in the data stored by an specific set of services within the OpenDaylight's REST API. The first of these services is the *Inventory Manager*. By accessing the route that can be seen in Listing 5.1, we can obtain data related to any routing element within the network. This is done by using the route specified in the listing and adding *node/{node identifier}*, where {*node identifier*} represents the identifier of the network node being consulted. Usually, this identifier consists in a name with the structure *openflow:n* where n represents an integer.

However, this would return a massive amount of data, most of it useless. The OpenFlow protocol requires that each node has a capacity of 255 routing tables. It also requires for every table to be instantiated on start, even though only one (the first one, routing table 0) is used to hold the routing rules sent by the controller. Therefore, if we just query all data regarding a node, we will obtain many empty fields. Therfore, we must be very selective regarding the data that we query regaring the Inventory Manager.

For example, if we want to obtain information of the routing rules being implemented in the *openflow:3* node (the third switch), we would need to send an HTTP GET query to the endpoint *http://localhost:8181/restconf/operational/opendaylight-inventory:nodes/node/openflow:3/flow-node-inventory:table/0*. The result of such query can be seen in Listing 5.2

This listing includes only part of the data obtained (since the complete document would contain many more fields regarding other flows), but here we can see the basics of the data representing routing rules. Fields showing statistics related to each flow and the routing table are of special interest. We are also interested in data showing the actual rules being implemented (in this case, the presented flow sends packets being received through port 1 to the controller and other nodes).

Listing 5.1: Query to the *Inventory Manager*

```
http://{Opendaylight Endpoint}/restconf/operational/opendaylight-inventory:nodes
```

Listing 5.2: Example of data obtained after querying the *Inventory Manager* for routing information

```
<table xmlns="urn:opendaylight:flow:inventory">
   <id>0</id>
   <flow-table-statistics xmlns="urn:opendaylight:flow:table:statistics">
      <active-flows>4</active-flows>
      <packets-looked-up>103</packets-looked-up>
      <packets-matched>100</packets-matched>
   </flow-table-statistics>
   <flow>
      <id>L2switch-1</id>
      <flow-statistics xmlns="urn:opendaylight:flow:statistics">
         <packet-count>33</packet-count>
         <duration>
            <nanosecond>449000000</nanosecond>
            <second>66</second>
         </duration>
         <byte-count>2590</byte-count>
      </flow-statistics>
      <priority>2</priority>
      <table_id>0</table_id>
      <cookie_mask>0</cookie_mask>
      <hard-timeout>0</hard-timeout>
      <match>
         <in-port>1</in-port>
      </match>
      <cookie>3098476543630901249</cookie>
      <flags />
      <instructions>
         <instruction>
            <order>0</order>
            <apply-actions>
               <action>
                  <order>1</order>
                  <output-action>
                     <max-length>65535</max-length>
                     <output-node-connector>CONTROLLER</output-node-connector>
                  </output-action>
               </action>
               <action>
                  <order>0</order>
                  <output-action>
                     <max-length>65535</max-length>
   (...)
```

We will also collect data from the *Topology Manager* in a similar manner. While the *Inventory Manager* holds data regarding the rules and statistics of each element within the network, the *Topology Manager* shows the topology and disposition of elements in the

virtualised network, both routing elements (links and nodes) and hosts.

These two OpenDaylight modules will be directly queried by Python scripts being executed periodically. Then, this data is going to be processed (this process will be explained in the next subsection) and both the original and processed versions of this data will be stored in the Elasticsearch module as ordered by the archtiecture of the Data Lake paradigm.

5.3.2 Data Processing

Once we have depicted the processes and tools used for the collection of data from the multiple sources already described, in this section we will describe the different processing steps being followed by each data flow.

First we are going to describe the steps involved in the processing of the data collected for the purpose of detecting the presence of attack patterns. As we already described in Subsection 5.3.1, this data is collected directly from the network. Once it reaches the Data Ingestion layer (through the use of Beats), the Logstash instance filters the data collected in order to separate data related to the detection of attacks from the rest of the data. Then, this data is sent to a submodule where the CICFlowMeter is used. This submodule is known as the Data Processing module.

The CICFlowMeter, which was already introduced in Subsection 2.7.3 plays a key role in the processing of this data. Since our intention while developing this use case was to replicate the scenario and environment described by the Canadian Institute of Cybersecurity involved in the generation of the data used in this thesis for the creation of the Machine Learning models, we have chosen use the CICFlowMeter tool (as they did) instead of developing our own processing system.

As we already mentioned, CICFlowMeter detects the presence of flows in PCAP files by analysing the contents of each message sequentially, and then it obtains multiple features regarding such flows. However, we are only interested in a small subset of those features. Specifically, we are interested in the features present in the Machine Learning model developed. These features are: Packet Length Mean, Avg Bwd Segment Size, Bwd Packet Length Max, Bwd Packet Length Std and Destination Port.

Since CICFLowMeter does not distinguish between packets but between flows, it is able to divide information extracted from such flows between information regarding data flowing in one direction (Forward, or Fwd) and data flowing in the opposite direction (Backward, or Bwd). In this case, we focus mostly in parameters related to data flowing from the server to the attacker.

Since, in order to detect flows, a set of sequential entries regarding traffic packets are necessary for the purpose of identifying the connection establishing and tearing processes, the data processing can not be done in real time, but in near-real time. Following this criteria, the CICFlowMeter is executed repeatedly over small subsets of data (representing captured packets) with a limited size.

As a result, we obtain a Comma Separated Value (CSV) file with 84 values. Even though we are only interested in the subset of features already mentioned, we will push all the data obtained into the Elasticsearch module. The Attack Detection submodule will take on the task of selecting a subset of features. This choice allows us to easily change the Machine Learning model being used and the features being used without modifying also the Data Processing module, since all the information generated by such module is already being provided, and the selection of features is performed within the Attack Detection submodule.

Regarding the data collected from the OpenDaylight controller, most of the processing steps performed within the Data Ingestion layer is limited to the formatting of the data in order to ease its injection into the Elasticsearch module. This data will be further processed; however, most of the processing steps related to such data consists in semantizing it. This process will be further described in Section 5.5.

However, we do perform some pre-processing regarding data collected from the collector. As we mentioned before, there is a quite large quantity of data fields that are instantiated (since the OpenFlow protocol requires that those data fields must be present in any database) but they do not hold any useful information. An example would be the 254 tables instantiated in the databases. These tables are completely empty, since all the routing information is stored in the first table. However, if we send queries to the controller in order to get data regarding routing entries in a node, we will get 254 entries with empty fields along with the data stored in the first table.

Therefore, we will perform some pre-processing consisting in deleting data fields instantiated but empty, such as the example previously described. This process is performed by implementing and executing a Python script which executes HTTP requests to OpenDaylight endpoints and uses an Elasticsearch client implemented by extending the Elasticsearch class provided by the Elasticsearch Python library for the injection of data. The same method is followed for the injection of data into Elasticsearch regarding the data processed by CICFlowMeter.

5.4 Cyber-attack Diagnosis

In this section we are going to describe the inner workings and implementation of the Attack Detection module. We will start by describing the methods followed in order to implement the model learned during the learning stage of the thesis using data provided by the Canadian Institute of Cybersecurity. The learning of this model is briefly introduced in Subsection 3.3.2. In this section we will further describe such model along with its implementation.

Furthermore, we will also describe the composing of rules regarding the blocking of detected attacks and the implementation of such methods. We will also depict the process followed for sending these rules to the OpenDaylight controller and how the controller accepts and implements the routing rules sent.

As we mentioned previously, the model selected for the detection of patterns related to DoS attacks is based on **Random Forest** Machine Learning models. These models are based on the concept of using multiple decision trees which are trained in a parallel manner with subsets selected with replacement from the original training dataset. Then, the resulting outputs of each tree are combined following a vote system where the output of each tree is weighted by the estimated probability provided by each tree along its output.

Even though other methods can be followed when combining the outputs of each tree that compose the Random Forest (such as non-weighted vote, regression, etc), this method provides the best results. Furthermore, it is implemented by the library used in order to programme the Machine Learning model.

However, first we are going to describe the process of collecting the processed data from the Elasticsearch module in order to perform predictions using the model previously mentioned. The output of each execution of the CICFlowMeter tool is pushed into an index named *csv_cicflowmeter*. In order to query the Elasticsearch module for data regarding this specific index, we build the data query that can be seen in Listing 5.3.As we can see, we included a *from_data* field in order to select data in a specific timeslot and avoid detecting the same attack multiple times. This query is included in a Python class which extends the ESClient class (provided by the Elasticsearch Python library) by implementing "get" and "set" methods that targets the *csv_cicflowmeter* specifically while also including variables such as the one previously mentioned. For each time that we query attack-related data, we update the from_date variable.

Listing 5.3: Time-based query used to extract attack-related data

```
{
    "query": {
      "range": {
        "@timestamp" : {
          "gt" : from_date
        }
      }
    },
    "size": 1000,
    "sort" : [
      {"@timestamp" : "desc"}
    ]
}
```

Once we have completed the data querying system, we focus on implementing the attack detection model. In this task, the use of the Scikit-learn library is key. As we already mentioned in Section 2.6, Scikit-Learn is the standard library used for the implementation of machine learning systems in Python.

However, first we have to extract the required features by the model. As we mentioned in the previous section, this process is implemented in the Attack Detection module instead of the Data Processing submodule in order to provide more flexibility to the overall architecture. Once the features have been selected, we use Pandas dataframes [38]. Pandas is one of the most used data managing libraries developed for Python, and it eases the selection and processing of batches of data given certain rules expressed with Python conditions.

Therefore, by building a dataframe object with the data retrieved from Elasticsearch and selecting only the necessary features, we have the data in the adequate format to predict attacks. Now we focus on implementing the previously learned model. This model was saved in a Pickle-generated file; Pickle allows us to store Python objects in regular files. Once we have loaded the model previously learnt (which is an instance of the RandomForestClassifier class within the Scikit-learn library), we iterate over each entry of the dataframe object, using each feature value in order to predict if that entry represents an attack.

Then, we collect the results in a new dataframe object, which holds all the features extracted by the Data Processing submodule for each entry classified as an attack. Thanks to this method, we can consult all the data generated by the CICFlowMeter for the entries classified as attacks. This is specially useful for the creation of rules in order to block such flows.

And, in fact, that is the next step after detecting a set of attacks. Once we have a

dataframe object that holds data regarding flows that are attacking our service, we use the Policies Creation submodule in order to develop routing rules following the Opendaylight data modelling structure so we can push them into the node providing access to the service through its northbound interface.

5.5 Semantic Data Enrichment

So far, we have described multiple processes involved in the collecting, "cleaning" and storing of data. These processes help us in adapting the data for its representation and the detection of attacks. However, none of the processes described focuses on standardising data by creating structures so they can be used in other contexts and new services can be developed over it. Specifically, we create ontologies that we instantiate with data currently stored in the Elasticsearch module using said ontologies and Python scripts.

Regarding the ontologies being used in this use case, we have already introduced them in Subsection 3.2.3 we briefly introduced the ontology being used in this use case for the purpose of representing both data regarding the general status of the network and specific fields showing the presence of attacks and its features.

As we have previously mentioned, the SDN-AR ontology consists of a combination of two ontologies: the SDN-NDL ontology and the UCO ontology, adding new entities and relationships to adapt both ontologies among themselves and also to this use case. The first ontology focuses on structuring data obtained from OpenDaylight-based controllers. Specifically, it defines an structure based on entities and relationships that maps data provided by each module within OpenDaylight. In fact, many entities are defined with the purpose of reflecting certain data structures.

On the other hand, the UCO ontology is completely focused on the standardisation of datasets that represent attacks. Since it is not designed with an specific attack in mind, it is a very general ontology. Furthermore, this ontology has not been designed considering the SDN paradigm. Therefore, we have selected a specific set of entities and relationships useful for our use case, and we have applied them to SDN related concepts (such as SDN nodes) for the representation of data collected over attacks being detected.

We have merged these two ontologies into the SDN-AR ontology, which includes new entities and relationships for the purpose of uniting UCO and SDN-NDL, such as the relationships *attacksNode*, *suffersAttack* and *createsAttack* (which connect the SDN-NDL-based entity *HostNode* with the UCO-based entity *Attack*, for the purpose of identifying which hosts are being attacked by which hosts).

Listing 5.4: Semantic Cyber-attack Report


```
{
   "@context":{
      "sdl":"http://www.gsi.dit.upm.es/
          ontologies/sdl/ontology.xml#",
      "uco":"https://ebiquity.github.io/
          Unified-Cybersecurity-Ontology/
          uco_1_5.owl#",
      "att":"http://www.gsi.dit.upm.es/
          ontologies/sdn/att-sdn#",
      "topology":"http://bayesiansdn.
          cluster.gsi.dit.upm.es/
          simulationattack/topology/",
      "@base":"http://bayesiansdn.cluster.
          gsi.dit.upm.es/simulationattack/
          attacks/",
      "@vocab":"http://www.gsi.dit.upm.es/
          ontologies/sdn/att-sdn#",
      "timestamp":null,
      "schema":"http://schema.org/"
   },
   "@id":"attack2",
   "@type":"uco:Attack",
   "timeOfAttack":"2019-05-14T10:13:21.000
       Z",
   "attacksNode":{
      "@id":"topology:
          host_06_f8_e3_ed_ea_cd",
      "@type":"sdl:HostNode"
   },
   "hasAttacker":{
      "@id":"attacker2",
      "@type":"uco:Attacker"
   },
   "hasTakenCOA":{
      "@id":"coa2",
      "@type":"uco:CourseofAction"
   }
}
```


Listing 5.5: Semantic Topology Report


```
{
   "@context":{...},
   "ndl:hasInterface":[
      {
         "@id":"openflow1_4",
         "@type":"Interface",
         "nml:name":"s1-eth4",
         "configuration":"",
         "currentFeature":"ten-gb-fd
             copper",
         "currentSpeed":10000000,
         "describesInterface":{
            "@id":"topology:openflow1_4",
            "@type":"Interface"
         },
         "hardwareAddress":"7e:81:0e:00:d6
             :e5",
         "hasNetworkStats":{
            "@id":"openflow1_4#
                interfaceStats",
            "@type":"InterfaceStats",
            "bytesReceived":16363,
            "bytesTransmitted":226971,
            "packetsReceived":187,
            "packetsTransmitted":2965,
            "receiveCRCError":0,
            "receiveDrops":0,
            "receiveErrors":0,
            "receiveFrameError":0,
            "receiveOverRunError":0,
            "transmitDrops":0,
            "transmitErrors":0
         },
         "hasState":{
            "@id":"openflow1_4#
                interfaceState",
            "@type":"InterfaceState",
(...)
```


In Listings 5.4 and 5.5 we can observe instances of the proposed ontology which references the presence of an attack and some of its variables (such as attackers or actions taken) and the topology of the network being monitored (specifically, part of the data obtained referencing one of the interfaces of the node openflow:1).

The process of semantizing the data is implemented using Python. Specifically, first we

define the SDN-AR ontology using the RDF file format, which is the standard language used for the definition of data models within the Semantic Web Standards defined by the World Wide Web Consortium.

Using this file as the basis for the structure of each document, we collect data from the Elasticsearch module. Depending on the index being queried we collect data regarding attacks or the current status of the network. Therefore, first we collect data related to detected attacks and then we create Linked Data tuples such as the ones observed in Listing 5.4. After creating tuples storing information about attacks, we access the indices that store data concerting the current status of the network to create Linked Data tuples such as the ones observed in Listing 5.5.

5.6 Visualizing Data

So far, we have developed a prototype of a system that deploys a virtualized SDN network controlled by an OpenDaylight instance where we simulate a service being attacked in order to collect data and detect such attacks using a Machine Learning model previously learned. Then, we semantize both the data collected from the network and its controller and the data generated by the Attack Detection module.

As a result, we obtain an Elasticsearch instance with large amounts of data grouped in different indices. Even though Elasticsearch provides an interface where queries can be interpreted and results can be consulted using a standard web browser, we would like to offer a more visual representation of the current status of the network.

In order to accomplish this, we can choose between two options. The first option would be to use the Kibana tool. Kibana is the standard data visualizer used in deployments that involve storing data in Elasticsearch. Since both Elasticsearch and Kibana are integrated into the ELK stack (as a matter of fact, the E stands for Elasticsearch and the K stands for Kibana), we considered this option first.

Even though it has some considerable advantages (such as an easy integration with the Elasticsearch module being used as the central element in the Data Lake architecture), it has the disadvantage of not being "specific" enough regarding the data being depicted. Since we store most of the semantic data in Elasticsearch as instances in JSONLD format, we can not take advantage of the benefits provided by the semantization of such data. Kibana is only useful for providing an easier insight in raw data stored in Elasticsearch.

Therefore, we have chosen the second option, which consists in developing our own visualization module in order to portray specific information that we considered of special

interest. For this purpose, we will use the framework known as Sefarad (already introduced in Section 2.5.2) as the base for such module.

Sefarad makes use of Polymer in order to control the cycle of data monitoring and rendering within the visualization system. In order to portray the data collected, we use multiple types of charts from the Google Chart web service. Data is collected both directly from Elasticsearch and using SPARQL queries for semantic data.

However, as we have previously mentioned, Elasticsearch stores semantic tuples in JSONLD-defined documents. Hence, it does not support SPARQL-based queries over semantic data. This is the reason for adding a Fuseki-Jena module which receives and attends each SPARQL query sent by the visualization module.

Before implementing and starting the visualization module, we push each JSONLD generated in the system and stored in Elasticsearch into the Jena-Fuseki module. We do this by creating a database within Jena-Fuseki, and then using its HTTP API for pushing such JSONLD files. The reasoning module within Jena-Fuseki will create tuples representing the entities and relationships being stored in each JSONLD file.

While the system is being run, this database keeps growing with new Linked Data tuples representing new data being collected from the network as well as new attacks being diagnosed. Therefore, when we reload the visualization system (which is accessed through its HTTP endpoint), we can observe new data being collected from the underlying system.

The whole visualization module has been implemented using Javascript; since all the libraries and web services previously mentioned are available for Javascript, the selection of the programming language was a very short process. We also used Bower for managing packets being used in our module, which is run using the http-server command.

In Listing 5.6, we can observe an example of a query used for retrieving attacks being detected by our system. We can further tune this query in order to select subsets of attacks depending on ranges in values of each attribute, such as time or origin of the attack. The listed query is executed for the purpose of collecting data enumerating the attacks being detected in the network. The results of this query are then showed in charts as the one showed in Figure 5.4, and the results of the overall attack tab of the dashboard can be seen in Figure 5.5.

Listing 5.6: Example of SPARQL query to collect data about the detected attacks

```
BASE <http://bayesiansdn.cluster.gsi.dit.upm.es/simulationattack/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX uco: <https://ebiquity.github.io/Unified-Cybersecurity-Ontology/uco_1_5.owl#>
```

```
PREFIX att: <http://www.gsi.dit.upm.es/ontologies/sdn/att-sdn#>

SELECT ?id ?timeOfAttack ?origin ?target ?blockingIp
WHERE {
  ?id rdf:type uco:Attack .
  ?id att:timeOfAttack ?timeOfAttack .
  ?id att:hasAttacker ?attacker .
  ?attacker att:hostedBy ?origin .
  ?id att:attacksNode ?target .
  ?id att:hasTakenCOA ?coa .
  ?coa att:composedBy ?flow .
  ?flow att:hasMatch ?match .
  ?match att:ipAddress ?blockingIp
}
```

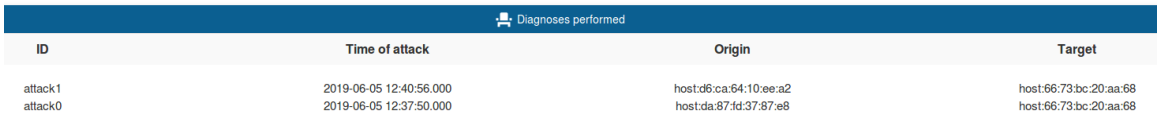

Figure 5.4: View of the attacks being detected

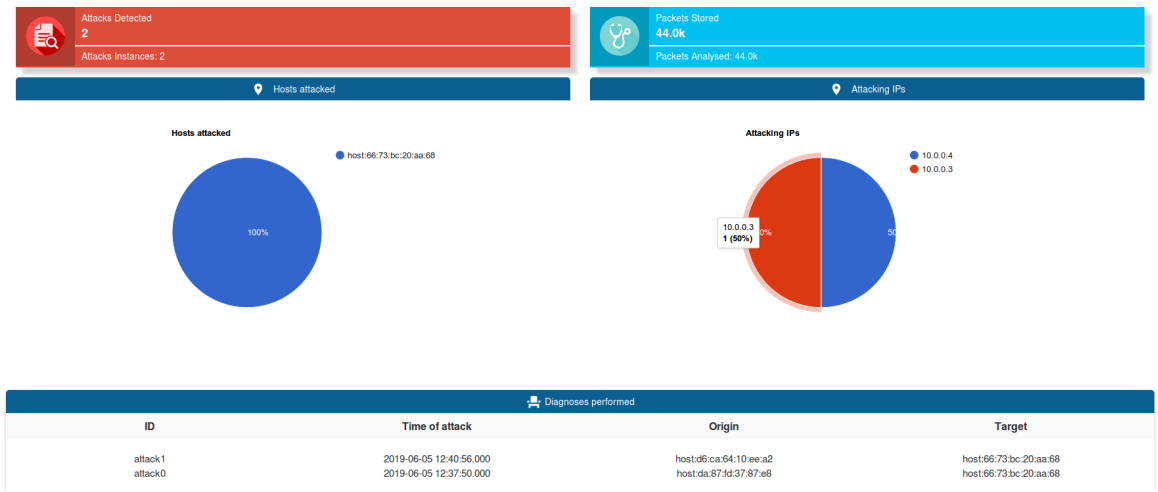

Figure 5.5: Overall view of attacks information

However, we do not use SPARQL queries just for collecting semantic data created by the Attack Detection module. We also use these queries to extract information held by the semantized version of the data collected from the OpenDaylight controller. This data, as we have previously mentioned, is collected for the purpose of obtaining an overview of the general status of the network and its nodes. We can observe an example of the representation of such overview in Figure 5.6.

Just as we used SPARQL queries for collecting data from the Jena-Fuseki instance and

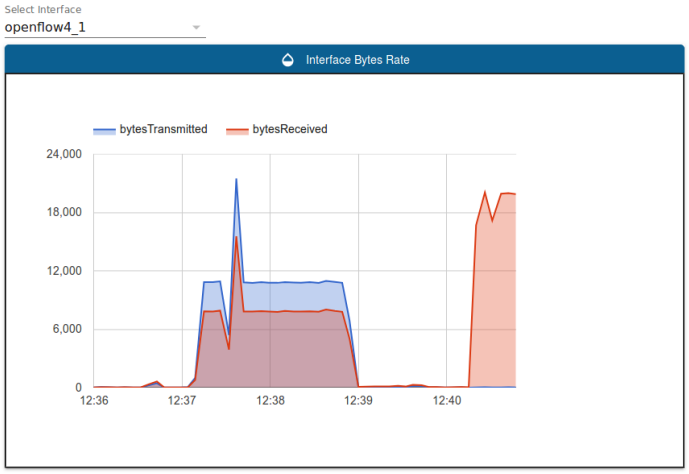

Figure 5.6: View of the data flow through an specific interface

then format this data for its representation using a chart, we have also followed the same process for other types of data. Specifically, we have followed the same process for such information as the number of anomalies in each interface of each node in the network, a counting of the number of attacks detected or the extraction of information regarding an specific node.

Furthermore, regarding SPARQL queries, we have also included a Sparql Editor where we can write our own queries. Then, this queries are sent directly to the Jena-Fuseki module, and the result of resolving such queries is sent back to the visualization module. Finally, the visualization system returns the result in a structure below the editor. In this view, which can be seen in Figure 5.8, there is also a drop-down menu where a set of pre-configured queries can be selected. These queries will be shown in the editor window once they are selected. Then, we can execute them directly or modify them to extract specific pieces of information.

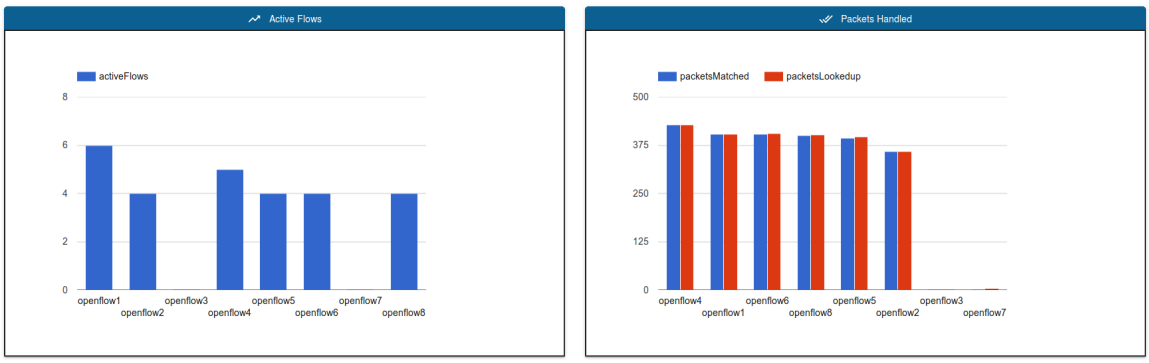

Figure 5.7: View of flows and packets handled by each node

However, we do not collect data only from the Jena-Fuseki module. The Sefarad framework allows us to collect data also from the Elasticsearch instance. For example, we use the Elasticsearch Javascript client in order to send a query to the Elasticsearch instance to collect information related to the JSON files representing the traffic being captured in the interfaces which connect the machine hosting the service to the network. Since this information is stored in an specific index within Elasticsearch, it is easy to collect such data. However, even if the structure of this information was complicated, it would only require a more complex query. Another example of data collected from Elasticsearch can be seen in Figure 5.7.

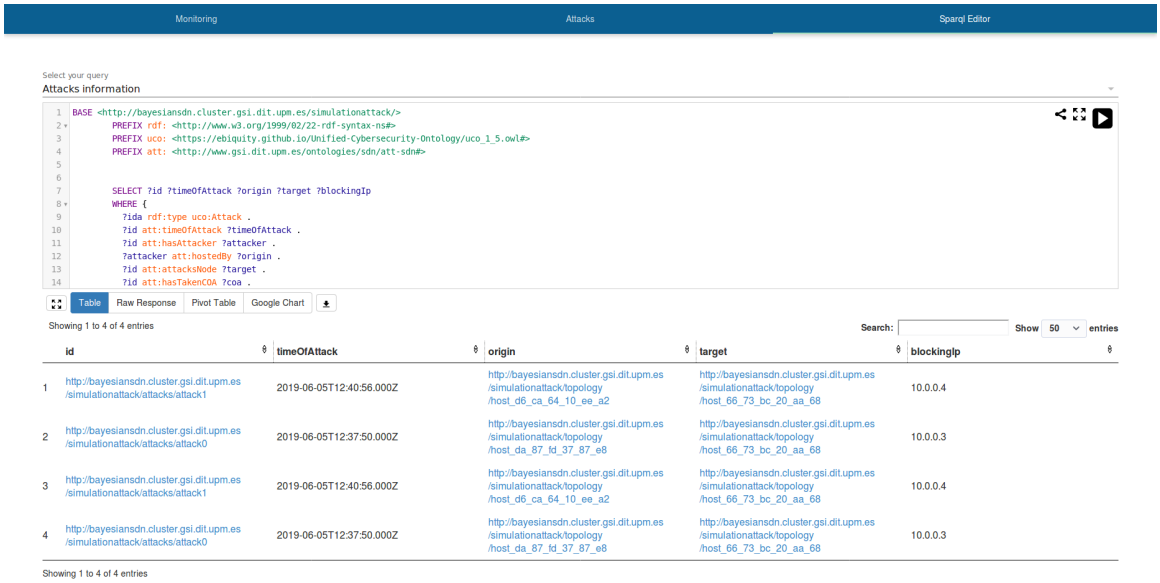

Figure 5.8: View of the Query Editor

CHAPTER 6

Conclusions and Future Work

This project has resulted in a system that provides a continuous and automated monitoring and detection of DoS attacks within a SDN context. Specifically, we have taken advantage of the benefits of the centralized management of the network that the SDN paradigm provides. We have also successfully developed a semantic model extended over other ontologies and used it to create tuples based on the collected data. Finally, we have also included a visualization system which eases the use of this system. However, this project can be further developed. In this chapter, we will deepen in the conclusions mentioned in this paragraph. We will also describe some paths for further improving this system in the future.

6.1 Conclusions

Our initial goal consisted in developing a system that takes advantage of the benefits provided by the centralized control of SDN networks for the purpose of detecting and blocking DoS attacks created in a virtualized SDN testbed has been achieved. Furthermore, we have also achieved the goal of implementing a semantic model that allowed the structuring of collected data. We have also developed a module for the semantization of raw and processed data.

Once we have selected and analyzed the tools to be used in the developing of the system, we enumerate the specific tasks needed for reaching the goals defined in the previous paragraph and divide these tasks among different elements, thus designing the architecture of this system. Then, we have applied this architecture to an specific use case, where we have used the tools previously selected to develop a deploy a scenario where an OpenDaylight-controlled SDN network used for providing a service is being attacked by a LOIC instance. The system deployed successfully detects this attack and pushes new rules into the controller for its blocking. Data regarding this scenario is stored in Elasticsearch, semantized, and provided to the user through a dashboard. This shows that the project has reached all the proposed goals.

Regarding conclusions drawn from this project, first we would like to emphasise the potential of the SDN paradigm. Due to the fact that the network control is centralized, we only needed to create one specific routing rule and push it into one endpoint to successfully block the detected attack without affecting other nodes or traffic flows.

Furthermore, we would like to highlight the potential use of Machine Learning models for the automating of tasks within SDN environments. These models allow for the treatment of ever-growing complex attack scenarios. However, a key role is played by the use of Big Data technologies, which are essential in providing tools and modules for the collecting and processing of large amounts of data needed for detecting and blocking the attacks previously mentioned.

Finally, we must also emphasise the opportunities given by current SDN-virtualizing environments and open-source controllers for developing systems focused on improving tasks in SDN without being able to access a real SDN network. The use of Mininet and OpenDaylight has allowed us to develop this project, even though we could not enjoy access to real SDN networks.

6.2 Future Work

This project has successfully solved the challenge of automating the detection and treatment of DoS scenarios within an SDN environment. However, we would like to extend the range of attack types being detected and processed by our system in order to apply it as a general cyberattack protection system for SDN environments.

Furthermore, we would like to extend the ontology being used in this project. By extending the current ontology, we could not only add new variables and data fields representing the current status of the network, but also represent complex attack scenarios where attacks do not have such a direct effect in the network being monitored. Furthermore, we would also like to develop entities to represent the services being provided.

As a matter of fact, we would like to focus also on including multiple services and more complex scenarios, where services depend on each other and we have to monitor multiple types of resources.

Finally, we would like to modify the data collection module in such a way that it can collect data from multiple types of SDN controllers, instead of being adapted to work with OpenDaylight controllers. For example, we would like to include methods to collect data from the ONOS controller, which is being widely supported by such telecommuncation companies as NTT and AT&T.

Bibliography

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[2] Akamai Technologies. The cost of denial-of-services attacks. `https://www.akamai.com/us/en/multimedia/documents/content/ponemon-institute-the-cost-of-ddos-attacks-white-paper.pdf`, 2015. Accessed: 2019-06-06.

[3] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[4] Cooperson, Dana and Chappell, Caroline. Telefonica's unica architecture strategy for network virtualisation. *Telefonica White Papers*, 2017.

[5] Orange Business Services. How sdn simplifies managing digital experiences. `https://www.orange-business.com/en/blogs/connecting-technology/networks/how-sdn-simplifies-managing-digital-experiences`, 2016. Accessed: 2019-06-06.

[6] Vodafone Business. Vodafone is first to launch new cisco sd-wan technology in europe. `https://www.vodafone.com/business/news-and-insights/white-paper/vodafone-is-first-to-launch-new-cisco-sd-wan-technology-in-europe`. Accessed: 2019-06-06.

[7] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[8] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using openflow: A survey. *IEEE communications surveys & tutorials*, 16(1):493–512, 2014.

[9] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 117–130, 2015.

[10] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014.

[11] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[12] Manda Sai Divya and Shiv Kumar Goyal. Elasticsearch: An advanced and quick search technique to handle voluminous data. *Compusoft*, 2(6):171, 2013.

[13] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*, page 17, 2012.

[14] Tom Heath and Christian Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[15] Dörthe Arndt, Ruben Verborgh, Jos De Roo, Hong Sun, Erik Mannens, and Rik Van de Walle. Semantics of notation3 logic: A solution for implicit quantification. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 127–143. Springer, 2015.

[16] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Rdf 1.1 turtle. *World Wide Web Consortium*, 2014.

[17] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. Json-ld 1.0. *W3C Recommendation*, 16:41, 2014.

[18] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[19] Ritika Bansal and Sonal Chawla. An approach for semantic information retrieval from ontology in computer science domain. *International Journal of Engineering and Advanced Technology (IJEAT)*, 4(2), 2014.

[20] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.

[21] Swizec Teller. *Data Visualization with d3. js.* Packt Publishing Ltd, 2013.

[22] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[23] Eli Bressert. *SciPy and NumPy: an overview for developers.* " O'Reilly Media, Inc.", 2012.

[24] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, pages 108–116, 2018.

[25] Arash Habibi Lashkari, Gerard Draper-Gil, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of tor traffic using time based features. In *ICISSP*, pages 253–262, 2017.

[26] Rajat Kandoi and Markku Antikainen. Denial-of-service attacks in openflow sdn networks. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 1322–1326. IEEE, 2015.

[27] Fernando Benayas, Álvaro Carrera, Manuel García-Amado, and Carlos A Iglesias. A semantic data lake framework for autonomous fault management in sdn environments. *Transactions on Emerging Telecommunications Technologies*, page e3629, 2019.

[28] Rob Enns. Netconf configuration protocol. Technical report, 2006.

[29] Zareen Syed, Ankur Padia, Tim Finin, Lisa Mathews, and Anupam Joshi. Uco: A unified cybersecurity ontology. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[30] Jeffrey Undercofer, Anupam Joshi, Tim Finin, John Pinkston, et al. A target-centric ontology for intrusion detection. In *Workshop on Ontologies in Distributed Systems, held at The 18th International Joint Conference on Artificial Intelligence*, 2003.

[31] Richard Lippmann, Joshua W Haines, David J Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Computer networks*, 34(4):579–595, 2000.

[32] Ciza Thomas, Vishwas Sharma, and N Balakrishnan. Usefulness of darpa dataset for intrusion detection system evaluation. In *Data Mining, Intrusion Detection, Information Assurance, and Data Networks Security 2008*, volume 6973, page 69730G. International Society for Optics and Photonics, 2008.

[33] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374, 2012.

[34] Jason King and Mark Easton. *Cross-platform. NET Development: Using Mono, Portable. NET, and Microsoft. NET*. Apress, 2004.

[35] Sukhveer Kaur, Japinder Singh, and Navtej Singh Ghumman. Network programmability using pox controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, volume 138, 2014.

[36] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[37] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[38] Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14, 2011.

[39] Ben Pfaff and Bruce Davie. The open vswitch database management protocol. Technical report, 2013.

[40] Kenneth J Pickering. Evaluating the viability of intrusion detection system benchmarking. *Bachelor Thesis, University of Virginia, US*, 2002.

[41] FRANCISCO DE and TORRE DÍAZ. La tributación del software en el irnr. algunos aspectos conflictivos. *Cuadernos de Formación. Colaboración*, 22(10), 2010.

APPENDIX A

Impact of the project

While usually services are the elements that generate the most value for companies, telecommunication networks are vital for the provisioning of such services. Therefore, the development of software systems on top of these networks that assure the correct functioning of such networks have great value for companies depending on these services. Furthermore, when these services are essential (such as services that support the management of emergency situations), they might become the target of attacks by cybercriminals that want to cause mayhem. However, protecting networks and monitoring their functioning can be very costly when using workforce. Therefore, the automating of such processes can result in a considerable benefit for companies managing these networks.

In this appendix the social, economic and environmental consequences of this project will be presented in Sections A.1, A.2 and A.3, respectively. A dissertation regarding its ethical consequences will also be included in Section A.4.

A.1 Social Impact

The social aspects related to this project are quite varied, but they can be grouped into two fields: those related to the assurance of service providing and those related to cybersecurity.

The use of multiple services through computer networks (such as streaming, transport or job-seeking services) have an ever-growing impact in our everyday life. Therefore, any event that hampers the ability of users to access those services will have an impact in their lives. This impact could be very significant if the services being affected are essential, such as emergency services. However, our system helps in avoiding situations in which services become inaccessible due to cyberattacks, hence reducing the impact of such failures in both minor and serious situations.

Furthermore, systems that discourage performing attacks on networks also have an impact on reducing both risks for people using services being provided by that network and the motivation of potential attackers for employing their resources on preparing themselves to perform an attack against a network. This disencouragement results in many people avoiding going out of their way to become attackers, since the resources and time needed to perform an attack successfully are now higher.

Finally, we must also remark the possibility of the creation of new jobs related to the deployment and maintaining of systems like the one developed in this project, and the creation of jobs related to the extension and further developing of the mentioned system.

A.2 Economic Impact

The automation of cyberattack protection-related tasks provided by this system could have a significant impact in companies that manage computer networks. Specifically, it could drastically reduce the resources consumed in the protection of the network. This protection could be guaranteed by a whole department tasked with providing cybersecurity; by automating their tasks, the resources consumed can be drastically reduced, since after the automation the company would just need to employ someone to maintain such automation. Another reason to expect a considerable impact as a consequence of the deployment of this system consists in reducing the mentioned value of 1,5 million US dollars mentioned in the Motivation section.

Furthermore, since we developed this system specifically for SDN networks, and these networks are being widely adopted by most telecommunication companies, the impact of this project will be amplified.

A.3 Environmental Impact

As we have mentioned in previous sections, computer networks are key for managing many different services and situations. This includes infrastructures which could have a heavy impact in the environment, such as power stations, mines or centers for the treatment of toxic waste. If any cyberattack were successful in making any service within that environment unavailable, the situation could rapidly evolve into an environmental catastrophe which could pollute significantly the surrounding areas and affect the lives of many people that lives near such places.

Regarding the specific prototype presented in this project, its development process had barely any effect on the environment. Since the testbed was virtualized using the tools already described, we did not make use of any devices bought specifically for this project. Therefore, we did not generated any waste during the development and testing of the system.

A.4 Ethical and Professional Implications

The main ethical implication related to the system developed consists in the collection of data from the network. Specifically, by collecting data that shows the details of each packet in the interface directly connected to the service, we should be very careful with the storing and treatment of this data so it stays anonymous and no profiles can be inferred from it. For this purpose, we should follow current laws regarding the treatment and privatization of data regarding traffic flows.

We also collect sensitive data regarding the structure of the network being monitored. A potential attacker could steal that data and obtain an insightful view of the topology and elements of the network. This data could help him/her in detecting weak points in the network topology, which could lead to a successful attack. As we stated in the previous paragraph, we should be very careful regarding the storing and treatment of data.

APPENDIX B

Cost of the System

The development of this project has required the expertise and time of the developer tasked with the mentioned project; therefore, we must consider his/her salary. Furthermore, we required computing elements for the development and testing of the system; these elements also have a cost that must be included in the overall cost of the system. If the resulting systems were to be commercialized, we should also include the expenditures related to the infraestructure needed for the deployment of such system. All the mentioned costs have associated taxes which must be considered.

Therefore, in this appendix we will enumerate the costs related to the development of the system. We will also provide a projection regarding the potential costs of deploying this system at a commercial level. First, in Section B.1, we will describe the elements required for the development of this project. Next, in Section B.2, we will depict the costs related to the workforce needed for the development and maintenance of this system. Finally, in Section B.3, we will describe the taxes related to these costs.

B.1 Physical Resources

First, in order to develop and test the system, a computer is needed. Furthermore, this computer must have enough memory and power for it to be able to run multiple containers at the same time. Therefore, we consider that this computing unit must provide the following resources:

- **Hard Disk**: 100 GB
- **RAM**: 16 GB
- **CPU**: Intel i7 processor, 2.80 GHz

We estimate the cost of a machine with these capabilities to be 800 €. During the development of this project, a machine with similar characteristics have been used. Then, this system has been deployed in a cluster consisting in a cluster head with a cost of 10,000 € and three computing units with a cost of 5,000 € each. This cluster also has a Network-attached Storage unit with a cost of 5,000 €.

However, the costs previously described apply only to the development and testing processes. If we intend to deploy the system at a professional scale, we should invest in a cluster of, minimum, seven computers. The cost of such deployment would be around 15,000 €.

B.2 Human Resources

Human resources are required in this project for both the development and maintenance of the system developed. We will estimate the salaries of the personnel required and the time required for each task.

The cost in working hours spent in the development of this project is estimated to be around 460 hours. This forecast is calculated considering that the academic programme estimates the cost of this project to be around 12 ECTS (European Credit Transfer System) credits. Since each credit entails from 25 to 30 work hours, the resulting time spent should be 360 hours; we added 100 hours spent in the documentation related to this project. Since the salary of an IT engineer could be considered to be around 2,500 € (gross) and a month has 23 working days, the development of this project should take three months at most (considering an eight-hour working day), or 7,500 € in costs related to workforce.

We must also consider the costs related to the maintenance of this project. In order to perform an optimal maintenance, we would need another IT engineer working full-time for the purpose of providing a quick response when an incident occurs. Therefore, we should consider the cost of employing him full-time, which will totals 30,000 € per year.

B.3 Taxes

Since we intend to further develop and commercialize this software, we must consider taxes related to the selling and buying of services and software systems in Spanish soil.

According to [41], a tax of the 15% of the product value is applied over transactions involving the mentioned product. However, this is applied only in Spanish soil: if we intended to commercialize this product in other countries, we should consider taxes applied in the country where we are selling our system.