POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

A transparent and secure gateway solution for IoT networks

Supervisor
prof. Antonio Lioy
dr. Daniele Canavese
dr. Leonardo Regano

Candidate
Lorenzo Cesetti

October 2023

Contents

Chapter 1

Introduction

Internet has been a revolutionary invention, maybe the most important, significant and life-changing for the whole humanity in the last decades. The invention, design and subsequent realization of a global computer network has made it possible the information and service sharing across the whole world and access to them by users in less than a few seconds. According to International Telecommunication Union (ITU)[1], "over the past three decades, the number of Internet users went from a few million in 1992 to almost five billion in 2021".

As well as the number of users, in recent years, the number of devices connected to the Internet has enormously increased [1]. All these connected devices are able to interact with the external world and with human beings thanks to a wide range of sensors that are able to convert analogical information read from the surrounding environment into digital data, through the digitization of some parameters of interest. This kind of devices can provide an enormous amount of data, and all this data is then communicated on the network, sharing them with other devices and with different applications and infrastructures. This modern category of devices enable an innovative paradigm, namely the Internet of Things (IoT).

IoT presents itself as a revolutionary paradigm. It provides the connection of an enormous number of devices to Internet and enables seamless communication and interaction between them. From the so called CIoT (Consumer IoT), category which include smart home and wearable devices such as smart thermostat and watches, to IIoT (Industrial IoT), which includes industrial sensors and autonomous machines, the proliferation of IoT devices has ushered in a new era of connectivity and services [2].

Therefore, different kinds of IoT devices operate in diverse contexts, from homes and offices to industrial supply chains. The IoT world consists of an expansive set of interconnected devices, each one is designed to carry out specific tasks and characterized by is ability to collect, elaborate, process and transmit large amounts of, potentially sensitive, data. With this respect, an example could be the exact location of a user in a specific moment or some measures taken from an industrial environment.

During the rapid expansion of IoT, which has introduced and still is introducing unprecedented possibilities and services, the presence of a fundamental challenge to be faced has come to the forefront, consisting in the intrinsic lack of security of IoT devices. In fact, while the massive interconnection introduced by IoT brings brand new possibilities, it also introduces several security vulnerabilities that malicious actors can exploit. By way of example, in 2014, security experts demonstrated they could hack a leading brand of network-connected light bulb and obtain the WiFi username and password of the household to which the light bulb was connected[2].

---

[1] https://www.itu.int/itu-d/reports/statistics/2022/05/30/gcr-foreword/ [Accessed: Jun 10, 2023]

[2] https://www.trendmicro.com/vinfo/pl/security/news/internet-of-things/smart-lightbulb-hack-lets-others-steal-your-wi-fi-password [Accessed: Jun 10, 2023]

In this context, it is due to cite Mirai malware, discovered in 2016 by the malware research group "MalwareMustDie"[3]. Mirai consists of a powerful malicious software that is able to turn IoT devices running Linux operating system into remotely controlled bots, which can be used as a botnet[4], increasing the offensive capabilities of a potential attacker especially when the objective is to perform a DoS[5] attack.

Therefore, IoT devices are often susceptible to an array of security flaws. Most common security issues that affect IoT devices consist in the usage of weak or hardcoded passwords, inadequate encryption mechanisms, and insecure communication protocols. Moreover, it is worth remarking that devices often lack mechanisms for timely security updates and patches, leaving them exposed when new vulnerabilities are discovered. The exploitation of these vulnerabilities can potentially conduct to consequences ranging from the disclosure of sensitive user data to even the control gaining over critical infrastructure systems.

These problems born from the fact that, to enable the necessary massive geographical presence of IoT, the devices themselves are often low-cost, low-power and characterized by significant restrictions in both memory and computing power. In this context, it can be said that the rise of IoT popularity has been very rapid compared to the development of robust security measures for this kind of devices, leaving them susceptible to various security threats.

Unlike traditional devices, such as laptops or smartphones, IoT devices are typically resource-constrained and, by design, they come with hardware on board which is limited in computing power and memory. Contextually, security-related aspects are often overlooked during the design and production stages, resulting in inherited vulnerabilities that are later challenging to mitigate.

This lack of attention in security-related aspects is influenced by several factors, which contribute to the intrinsic lack of security in IoT devices. First, it is due to mention that manufacturers are often required to enter products in the market quickly and at the lowest possible price, often sacrificing security measures in the process, which can introduce additional costs.

In addition, the presence of a wide range of diverse devices and manufacturers makes it harder to design uniform security standards and practices. This lack of standardization further puts a strain on the security, making it challenging to design and consequently enforce consistent security measures and protocols within the IoT ecosystem.

The ramifications of the intrinsic lack of security in IoT devices are far-reaching. Thanks to the interconnected nature of IoT devices, a single compromised device can serve as a gateway to a malicious actor, introducing the possibility to infiltrate an entire network or system. In the context of critical infrastructures, this kind of security flaws can have destructive consequences, such as disrupting services causing devastating effects potentially also affecting human beings, such as when considering healthcare systems, and causing significant economic losses.

As the number of IoT devices continues to grow exponentially, while the design of security mechanisms and facilities for these devices trudges, the probability for large-scale security incidents always amplifies, introducing and underlining the urgent need for robust security measures. In this scenario, the present work, started as part of the FISHY[6] project, which received funding from the European Union's Horizon 2020 research and innovation program, represents a viable way to address IoT devices-related security issues.

In particular, FISHY aims to create a coordinated cyber-resilient platform oriented to the establishing of trusted supply chains. The platform is based on novel evidence-based security

[3] https://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html [Accessed: Jun 12, 2023]

[4] A botnet consists of a network of bots, which in this context means malware infected devices, all acting under the control of a single malicious system.

[5] Denial of Service: is a type of DoS attack which aims to block the normal traffic of a victim, which can be represented by a single server, service, or a whole network, overloading the victim by flooding it with network traffic.

[6] https://fishy-project.eu/ [Accessed: May 28, 2023]

assurance methodologies and metrics, leveraging state-of-the-art solutions, such as virtualization of network functions, with the objective of enhancing the security of the supply chain.

In particular, the present work consisted of the realization of one FISHY platform component, namely IoT Proxy. The idea at the foundation in the realization of IoT Proxy consists of improving the overall IoT devices security through the externalization of security-related aspects of this kind of devices.

Practically, this idea is implemented forcing all traffic coming from and/or directed to IoT devices to pass through a secure network gateway, which consists in a more powerful machine, where a cascade of security controls is provided and actual controls are performed. In this way, the security of the single IoT device is increased and, in addition, each IoT device is "isolated" from the networking point of view, thus reducing the possibility, in case a single device was compromised, of threat spreading.

IoT Proxy consists of a modular secure gateway for IoT devices networks and practically implements the aforementioned ideas. Security controls are implemented by means of IoT Proxy modules that can be switched on and off and configured at need, through the configuration of IoT Proxy itself. With this respect, IoT Proxy offers the possibility to be configured in two different ways.

In fact, IoT Proxy can be used as a standalone solution or as part of the FISHY platform. When used as a standalone product, IoT Proxy exposes a predefined set of APIs which can be contacted to send configuration commands. On the other hand, when IoT Proxy is used as part of the FISHY platform, it can be configured publishing messages containing configuration commands on an message queue available on the web.

IoT Proxy and all its modules are virtualized products, which are fully delivered in the form of software. The fact of being virtualized provides the possibility to execute IoT Proxy within on a general purpose machine, potentially introducing the opportunity of shaping the hardware resources in relation to the specific expected workload, providing the property of scalability to the product.

As said, IoT Proxy is modular, which means it supports potentially many VNFs. In particular, the version of IoT Proxy delivered in the context of this work, already supports two modules. In this context, it is due to mention that all IoT Proxy modules are completely "transparent" to IoT devices, which means that no actions are required on the devices themselves, but the modules offer "plug-and-play" functionalities.

In particular, one of the supported modules was designed from scratch in the context of this work. This particular module consists of a custom Intrusion Prevention System (IPS)[7]. The provided IPS is based on machine learning techniques, in particular on a classification model, and it able to recognize, through the analysis of the traffic inside the network, two different types of attacks which are particularly relevant and effective with respect to the IoT world.

The IPS module also implements the so called concept of "oblivious authentication". In cybersecurity, authentication consists in the process of verifying the identity of a user, a process, a device, or in general a communication actor, and traditionally is accomplished by demonstration of possession of a secret by the actor. Oblivious authentication follows a different approach which is based on the usage of machine learning techniques and aims to authenticate an entity, in this case an IoT device, analyzing the network traffic profile of the device itself. Through this analysis, it could be possible to identify when the characteristics of traffic related to a given IoT device deviates from the typical traffic profile expected for it. In this way, an alert can be generated in presence of anomalous situations or events, with the purpose of promptly take action and mitigate the eventual threat.

Additionally, as said, another module is supported by IoT Proxy, which consists in a vNSF

[7]IPS is a network security tool function that monitors a network for eventual malicious activity, observing the network traffic, and in case takes action to prevent it, such as dropping the related traffic.

executing strongSwan[8]. strongSwan consists of a complete VPN[9] solution and provides the possibility to communicate over the public Internet as it is done in private networks, also introducing encryption and authentication functionalities to servers and clients, through the creation of IPsec [3] tunnels transparent to connected devices.

The present work illustrates the design and implementation process of IoT Proxy and its associated IPS module, providing also a discussion about the insights, ideas and technologies which have made possible the realization of the proposed solution.

In particular, Chapter 2 discusses about virtual Network Security Functions (vNSFs), investigating the concept of network function virtualization and presenting the design standards defined in this context. Then, Chapter 3 consists of a discussion about virtualization, which presents the functioning of various virtualization techniques and the concept of lightweight virtualization, introducing also modern solutions available in this context. After, Chapter 4 presents the concept of machine learning and provides an explanation of how this kind of technologies work, with particular focus on classification problems, also providing the description of some relevant techniques. Chapter 5 deepens the cybersecurity-related relevant aspects in the context of the proposed solution, IPS and VPN gateway, with particular focus on the advantages introduced by virtualizing such components and on the threats mitigated by using them. Chapter 6 describes the design and implementation of the proposed solution in the context of this work, deepening the original idea behind it and explaining the implementation process of IoT Proxy and IPS module. Consequently, Chapter 7 shows experimental results related to the performances of the proposed solution and Chapter 8 indicates related works. Finally, Chapter 9 contains the conclusions that were drawn at the end of this work and indicates some future works.

[8] https://www.strongswan.org/ [Accessed: Jun 10, 2023]

[9] A Virtual Private Network, or VPN, is an encrypted connection over the Internet from a device to a network, from a network to another network or from a device to another device.

Chapter 2

Virtual Network Security Functions (vNSFs)

Network Functions (NFs) are basic building blocks within a network infrastructure, characterized by well-defined external interfaces and well-defined functional behaviors [4]. Network routing, such as Domain Name Service (DNS) [5] or Network Address Translation (NAT) [6] services are examples of network functions. Someway, the network function needs to be implemented to provide its functionality to the network. Initially, the way to go was to build specific hardware equipment to implement each network function.

Considering the vast number of different network functions, which are much higher in number than the ones mentioned before, and the huge dissimilarity occurring in both external interfaces and functional behavior, originally there was the need of a big number of different specific hardware to correctly implement the needed network functions and to be compliant with the market demand. Each different network function was instantiated in a different hardware profile, capable to implement the specific functional behavior for the given network function, introducing heavy dependence on specialized hardware.

Originally, telecommunication industry was based on network operators deploying physical equipment for each function that is part of a specific service. Moreover, it is due to cite that service components have strict chaining that must be preserved in the network topology. This, combined with the fact that these products must submit to requirements for high quality, stability and stringent protocol adherence, implied very long production cycles, very low agility and, as said above, strict dependence on specific hardware [7].

In addition, the need coming from users for high data-rate services continued to grow. In this scenario, Telecommunication Service Providers (TSPs) were obliged to continuously purchase and operate new physical equipment. All of this led to significant cost increases for these companies. Furthermore, TSPs have experienced that an increase in costs and demand does not reflects in higher subscription fees because of the fierce competition, both among themselves and from services working on the top of their channels [7].

Therefore, the problem was the need of a large number of different specific hardware appliances in order to dispose of all the needed network functionalities. In this situation, TSPs were obliged to find solutions to cut expenses and build more service-oriented and agile infrastructures.

European Telecommunication Standard Institute (ETSI) took stock of this situation in October 2012, publishing a "call for action" document during the "SDN and OpenFlow World Congress", in Darmstadt (Germany) [8]. Summarizing the content of the document, ETSI basically highlighted three major problems:

- network operators' networks were populated by a large quantity of proprietary hardware appliances, leading to the problem that to launch a new network service usually required finding the space and power to accommodate all those boxes, which was becoming gradually more challenging;

- increasing cost of the energy, capital investment challenges and rarity of necessary skills to design, integrate and operate increasingly complex hardware-based appliances;
- shortness of hardware-based appliances lifecycles, which required a significant part of the whole procure-design-integrate-deploy cycle to be repeated with little or no revenue benefit.

In this scenario, virtualization born as a solution to these problems, allowing to decouple the NF from the actual hardware it runs on. It is due to mention that VNFs can be deployed in the provider network to deliver not only networking facilities, but also security services [9], allowing the operators to adopt security-oriented VNFs, which are known as Virtual Network Security Functions (vNSFs). It is also worth remarking that the general concept of decoupling NFs from dedicated hardware does not necessarily require virtualization of resources. However, in the modern world, the trend is indeed to move toward cloud technologies, making these NFs would run on commodity servers instead of on dedicated hardware. In such a way, the gains (such as flexibility, dynamic resource scaling, energy efficiency) anticipated from running these functions on virtualized resources are very strong selling points of virtualization Mijumbi2016.

With respect to this, it is worth remarking the companies massive migration to the cloud, as specified also by Eurostat: the statistics office of the European Union, which elaborates data coming from the member countries for statistics purposes. As shown by Eurostat [1], with regard to cloud computing usage statistics for years 2020 and 2021, use of cloud services by enterprises has grown for the vast majority of the countries across these two years, reaching in 2021 an EU average of 41% and presenting a peak of 75% for northern countries: Sweden and Finland. In addition, as specified in the 2021 edition of the regional yearbook by Eurostat [10], the European Commission presented a vision for the EU's digital transformation by 2030 which, among the other targets, specifies the objective of having 75% of EU enterprises making use of the cloud computing services.

2.1 Network Function Virtualization (NFV)

Exploiting the virtualization technology, NFV permits to explore new different ways for deploying network services. The basic idea behind NFV is to decouple the NF from the hardware it runs on. This ensures that a NF can be delivered to TSPs as an instance of plain software, allowing the consolidation of many different network equipment types into common off-the-shelf hardware, such as industry standard high-volume servers, switches and storage, which could be located and organized in structures such as data centers, network nodes and in the end user premises. With this solution, a network service could be organized in a set of Virtual Network Functions (VNFs), that can then be implemented in software and run onto market-standard devices. This scenario opens the doors to a set of possibilities and to a more agile way of delivering network services, also in different regions, since VNFs can be instantiated and relocated at many network locations, without the need to purchase and install new hardware equipment.

Obviously, even if virtualization techniques and other technologies are used, there is the need of underneath hardware, because after all, at the base there is always need of hardware to realize functionalities. For this trivial reason, from the practical point of view, to make all these concepts effectively usable and to really benefit from the advantages introduced by NFV, a whole architecture needs to be engineered.

In addition, it is due to cite that, even if NFV offers huge benefits, it also introduces differences in the way network services are realized and the necessity of managing them in new different ways. Moreover, it is not rare that some NFs are still deployed in a traditional non-virtualized way, because the migration to the NFV is still in progress or simply for choices at providers' discretion, thus introducing the necessity for interoperability between virtualized and non-virtualized network functions.

[1] https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises [Accessed: Jul. 16, 2022]

In October 2010, ETSI published its solution to support NFV technologies, proposing an architectural framework which defines the high-level functional architecture and design philosophy of VNFs and of the underlying infrastructure [11].

The aforesaid architecture has to focus on different aspects uniquely related to virtualization, whilst it cannot act on the points that are in common to physical and virtualized network functions. Summarizing what ETSI indicates in the same document, the architectural framework is in charge of:

- supporting VNF operations across different computing resources;
- defining a software architecture which uses VNFs as building blocks to construct VNF Forwarding Graphs (VNF-FG)[2], which outlines which VNFs are linked together and how they are linked in order to deploy a network service;
- interfacing management and orchestration of NFV with other management systems, like EMS (Element Management System);
- supporting different network services with different characteristics in terms of reliability and availability, exploiting virtualization techniques;
- assuring that virtualization does not introduce security risks and threats;
- facing virtualization-related performance issues;
- keeping minimal the impact introduced by the coexistence and collaboration of virtualized and non-virtualized NFs;
- leveraging existing technologies.

On the other hand, the following aspects are out of scope for the NFV architecture:

- the specifics of the NF itself, the adopted protocol and the management functions related to the functionality performed by the NF;
- direct control or management of the physical network;
- the packet flow, control, operations and management of the E2E (End-To-End) service. All these points should be independent of whether end points, NFs or underlying infrastructure are physical or virtualized;
- details in the implementation of the architecture itself.

Overall, NFV means deploy NFs as plain-software products and run them over a Network Function Virtualization Infrastructure (NFVI), managing them through an orchestration component. These three domains (VNF, NFVI and orchestration) are identified in the high-level architectural framework proposed by ETSI for NFV, represented in Figure 2.1:

- VNF: is the software implementation of a NF which can run over a NFVI;
- NFVI: includes physical hardware and defines how these resources are virtualized, to support VNFs execution.
- NFV orchestration and management: comprises orchestration and lifecycle management of hardware and/or software resources that make up the NFVI and the lifecycle handling of VNFs.

[2]The VNF-FG is a graph that shows VNF as nodes and logical links among them as edges. VNF-FG is intended to describe the traffic flow between the VNFs that compose it (e.g. VNF-FG for control traffic or VNF-FG for user traffic).

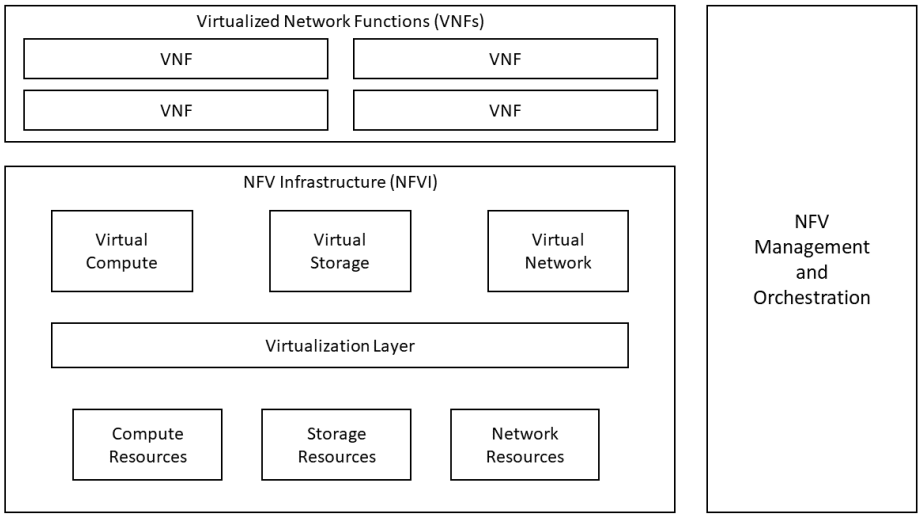

Figure 2.1. NFV reference architectural framework

ETSI definition of the NFV architecture is functional-level, thus not proposing any specific implementation. In particular, the NFV architectural framework covers aspects related to how the operator's network changes as a result of NFV process, focusing on new functional blocks carried by the virtualization and reference points between them. Some of these entities are conventionally present in traditional network deployments, whilst other may need to be added to support the virtualization operations.

A detailed view of the NFV architecture standard by ETSI is shown in Figure 2.2. In the picture, reference points[3] are represented by lines. Solid lines correspond to main and execution reference points and these fall under scope of NFV, since they are the targets of the standardization of the architecture. On the other hand, dotted lines represent reference points that are not treated mainly by NFV and correspond to those reference points that are already present in traditional non-virtualized network deployments, even if hey may need extensions to manage NFV. It is worth remarking that the proposed architectural framework does not indicate the specific NFs to be virtualized, since this is a provider solo decision, but it relates only to the functionalities that are necessary for the virtualization.

A VNF is, as already explained, a virtualized instance of a NF. What is important in this context is that the functional behavior, the internal state of the NF and the external operational interfaces must remain independent from the fact that the NF is virtualized or not, remaining the same in both cases. For each VNF, an EMS is provided, with the aim to collect, consolidate and normalizing data for and from the VNF associated VNF.

2.1.1 NFVI

Going deeply in the architecture, the NFVI is present. NFVI is the core component of the whole architecture and it represents the environment in which VNFs are implemented, managed and launched. This environment is composed by several hardware and software components, which provide the desired virtual resources to VNFs. When talking about hardware resources, means computing, storage and network, in order to provide processing power, storage and connectivity to VNFs, respectively, thanks to the virtualization layer. The latter is an abstraction of the hardware resources and is the component that makes it really possible to decouple the VNF software from the underlying hardware it runs on.

[3]In this context, a reference point means a point to point interface used to interconnect two different elements.

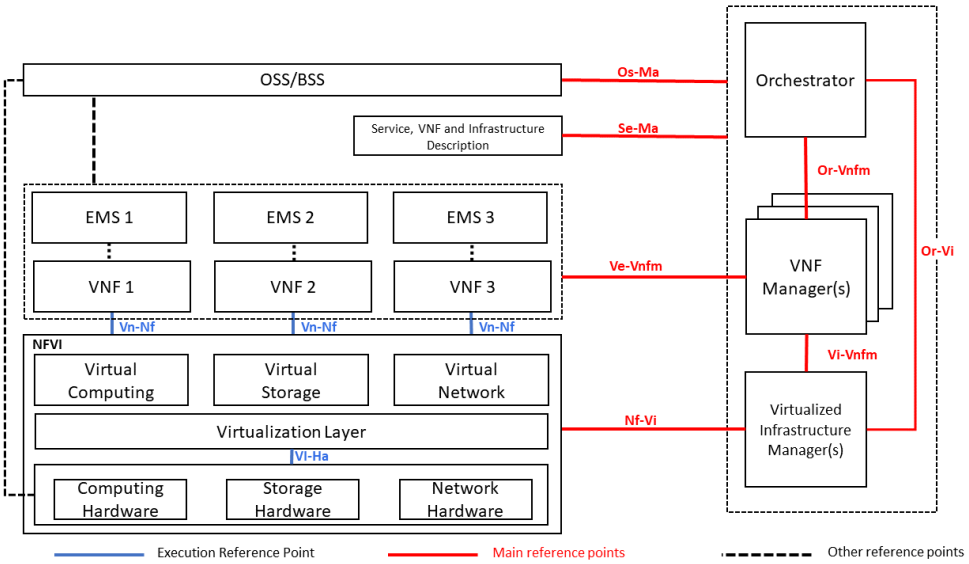

Figure 2.2. high-level NFV architectural framework

Summarizing, the virtualization layer is in charge of:

- abstract and logically divide physical resources, acting as a Hardware Abstraction Layer (HAL)[4];
- provide the possibility for the VNF to operate the underneath virtual infrastructure;
- assign to the VNF the virtualized resources.

Usually, these functionalities are provided by means of Virtual Machines (VMs) and hypervisors[5] for what concerns computing and storage. Hypervisors can act as HAL, virtualizing the available resources and assign them to different VMs, which offer an execution environment that runs on the virtualized hardware. Use of hypervisors is indeed the typical solution for the deployment of VNFs, although further solutions are present, like running VNF software directly on top of an operating system and physical hardware. The operations performed by the VNF must be independent of the deployment scenario, to guarantee operational transparency. When virtualization is used for what concerns the networking, the virtualization layer is responsible for abstract the network hardware and provide connectivity between different VNFs and to different VMs that compose a VNF, since a single VNF can also be deployed as a set of sub-components, each one running on a different VM. Most adopted solutions envisage Generic Routing Encapsulation (GRE)[6][12] and Virtual Local Area Network (VLAN)[7], even if the former is usually preferred since the latter entails having to act directly on the physical network settings, introducing dependencies on the physical network equipment.

[4]HAL is a software layer that allows to interact hardware devices at a general (or abstract) layer, introducing the possibility to bypass some device-specific aspects, like how the bits must be arranged in memory to communicate a certain command to the specific device, and other similar aspects.

[5]Hypervisor is the software that creates and manages virtual machines. It is also known as VMM (Virtual Machine Monitor).

[6]GRE is a protocol that allows to encapsulate information related to a set of different network layer protocols inside an IP packet.

[7]VLAN includes a series of technologies that allow to create different LANs that are independent each other, but that run over the same LAN physical infrastructure.

2.1.2 VIM

Beside the NFVI, the architectural framework presents the Virtualized Infrastructure Manager (VIM). VIM offers the set of functionalities that are needed to handle interactions of VNFs with their own computing, network and storage resources and is also responsible for resources virtualization. Firstly, it performs resource management, which means:

- keeping track of all the software and hardware resources that are assigned to the NFVI;
- allocation of virtualization components, like VMs on top of hypervisors;
- management of resources allocation and scaling (e.g. increase resources for a specific VM).

Furthermore, VIM is capable of performing a series of operations to provide visibility into and management of NFVI, as well as analysis operations like understanding the cause of performance problems or collection of NFVI faults information.

2.1.3 Orchestrator

Finally, Orchestrator component is in charge of the orchestration and management of NFVI and the related software resources, as well as realizing network services on top of NFVI. VNF Manager has the responsibility of handling VNF lifecycle, which means it dedicates to instantiate, update, query, scale and terminate VNFs. The component indicated as "Service, VNF and Infrastructure Description" is basically a data-set that provides relevant information about VNF deployment and VNF-FG, as well as service-related information. OSS/BSS (Operations Support Systems and Business Support Systems) are TSP-specific network components, responsible for operations state management and business-to-customer interfacing, respectively. Lastly, reference points are used by a components to communicate, exchanging control information each other; examples could be the request for resource allocation by the VNF manager to the VIM, which is mediated by the Vi-Vnfm reference point, or the state information exchange between Orchestrator and VIM, handled by the Or-Vi reference point [13].

2.2 Virtualized Network Functions (VNFs)

A VNF is a software product which, like a NF in a physical network, is a basic building block which runs into a NFVI, has well-defined external interfaces and a well-defined functional behavior. Developing a VNF could mean to write software that implements a NF defined by standardization organizations as IETF, following the specifications for interfaces and functional behavior, groups of network entities or to design a brand new NF presenting custom interfaces and behavior for specific purposes. By a way of illustration, respectively, a VNF could be a software product designed to virtualize DNS, as routing and NAT together or as a custom NF which could be useful for particular situations, like a VNF that implements a custom authentication mechanism.

A single VNF is usually composed by a set of VNFCs (Virtual Network Function Components). VNFCs are software sub-entities that are interfaced each other within the perimeter of the VNF they belong to and they are not visible from outside. Indeed, VNFCs are not restricted to follow specific and well-defined functional behavior or to communicate each other through well-defined interfaces, since they exist only inside the specific VNF. It is worth remarking that a VNF composed by a set of VNFCs is not necessarily implementing a group of NF, as one of the aforementioned types, but it only means that it consists of more software sub-modules which communicate each other to provide the full functionality offered by the VNF. Moreover, it is due to cite that how a VNF is structured and which are its sub-components is a provider-specific choice, which depends on many different factors, like scalability, security or other non-functional consideration, and a provider could also change a VNFC specifications, in terms of behavior and interfaces, over time, for example for a matter of performance. These factors further indicates the loss of well definition of functional behavior and external interfaces for VNFCs, which leads

to the fact that, in general, VNFCs cannot be assumed to have interfaces or behavior that are meaningful outside the enclosing VNF scope, although these components are responsible to expose the well-defined external interface for the VNF [14].

When there is the need to instantiate a VNF, the VNF Manager (presented in the section 2.1 as part of the NFVI) deploys a VNFI (VNF Instance) by deploying one or more VNFCIs (VNFC Instance), each one running inside its own virtualization container, whether it is a virtual machine or a operating system container[8]. In fact, a VNF is an abstract software entity that allows to define a sort of contract, specifying what a VNFI will do when it will be deployed at runtime. In the same way, VNFCs are TSP-specific software products of a given VNF, and VNFCIs are the executing parts that build up a VNFI and together provide the functionalities offered by the VNF.

Each VNF has always exactly one associated VNFD (VNF Descriptor), which describes the requirements for the initial deployment state and specifies also the functional and non-functional assurance that the VNF provides. These assurances are granted as long as the VNF's integrity is maintained, which means that the software components which constitute the VNF are the ones given by the provider. In fact, usually VNF are packaged by a single provider, which takes care of the development of all the VNFCs or, if it uses third-party software, has the responsibility to provide an accordant VNF package.

Before going deeply inside the structure of a VNF, for a further comprehension, it is due to remember that the objective of NFV, and therefore of VNFs, is to make possible to decouple network functions and capabilities from the infrastructure they run upon. This introduces several advantages and one of the most significant of them is the possibility to scale up and down the VNFs whenever it is necessary, to adapt the capacity of a VNF in a dynamic way. In order to meet these design goals, two factors need to be taken into consideration:

- functionality: a VNF has to provide commonly industry-defined or custom functionalities to the network they are deployed in;
- atomicity: a VNF needs to be atomic in order to ensure that its development, testing and deployment can take place independently from other VNFs.

VNFs communicates each other through interfaces, which are defined as points of interactions between VNFs. These reference points[9] are well-defined links that are exploited to exchange data and control commands between two entities, being them two VNFs or a VNF and another component. It is due to cite that these interfaces are not constrained to be implemented in software, but in the practice they could be also implemented in hardware, as any other component in the whole VNFI, like the VNF Manager. Focusing on the software-implemented VNF interfaces, the VNF software architecture defined by ETSI, defined in the document ETSI GS NFV-SWA 001, published in December 2014, presents the standard VNF interfaces, indicating them as the only point of contact for communication among VNFs.

As shown in Figure 2.3, exactly five different types of VNF's interfaces are defined:

- SWA-1: the SWA-1 interface permits communication by means of data and/or control plane among different VNFs that reside in the same network service or between VNFs that are part of different ones. A single VNF can support more than a single one SWA-1 interface. The VNFD must provide sufficient information about this SWA-1 interface;
- SWA-2: the SWA-2 interface is related to intra-VNF and consequently inter-VNFC communications. The implementation of these interfaces is vendor-specific. SWA-2 interfaces typically imply performance requirements, by means of latency or capacity, on the underlying NFVI;

[8] Virtual machines and OS containers will be presented in details in Chapter 3

[9] Interfaces are often referred as reference points in the ETSI nomenclature.

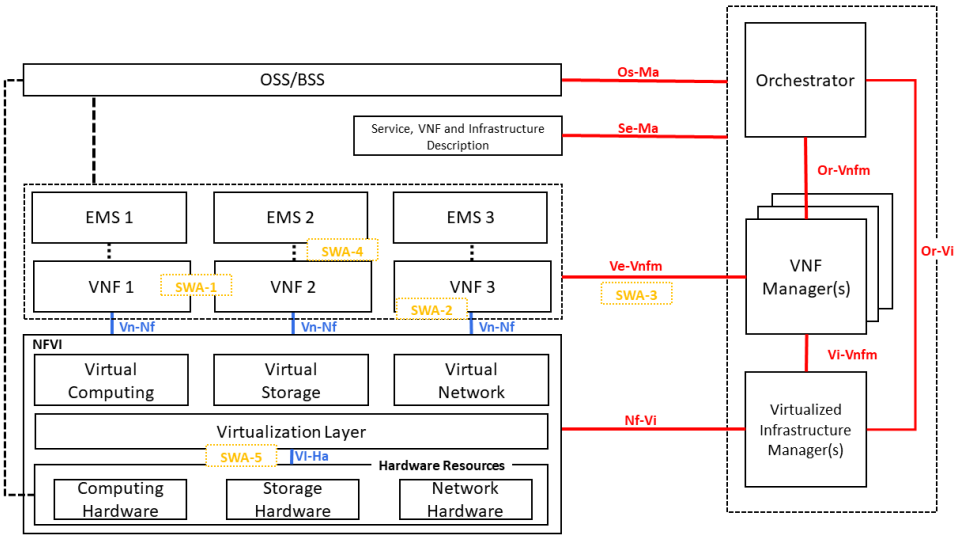

Figure 2.3. high-level NFV architectural framework with interfaces

- SWA-3: the SWA-3 interface is a point of interaction between the VNF and the VNF Manager. This type of interfaces are used for the communications of controls related to the creation, deployment, scaling, updating and destroying of a VNF, allowing the lifecycle management of it by the VNF Manager.

- SWA-4: the SWA-4 interface is the point of contact between a VNF and its associated EM. The objective of this interface is to allow the runtime management of the VNF, according to the FAB (Fulfillment Assurance Billing) network management model defined by the Business Process Framework (eTOM)[10], which is the standard operating model framework for TSPs maintained by the TM Forum[11].

- SWA-5: the SWA-5 interface consists in a set of interfaces that constitute the contact point between the VNF and the underlying NFVI and defines the execution environment for a VNF instance. To effectively deploy a VNF instance, it needs a set of resources, i.e. computing power, memory and storage. The needed subset of physical resources is virtualized by NFVI and provided in the form of virtual computing power, storage and network to the specific VNF instance through the SWA-5 interface.

Lastly, when talking about VNFs, it is due to cite aspects related to load balancing and scaling. If a VNF is parallelizable, namely if multiple instances of the same VNF can be instantiated at the same time, bringing advantages from the performance point of view, appropriate management of load balancing and scaling becomes crucial. ETSI, in its document defines some models for these two [14].

In this context, ETSI indicates that there are four load balancing models. A VNF can have an internal load balancer, which means that it is seen from outside as a single VNF, which inside has one or more VNFCs that are replicated and a load balancer is responsible to scatter packets or flows to one of them. If the VNFCs are stateful, than the load balancer has also the responsibility to redirect the traffic to the appropriate VNFC instance. Similarly, an external load balancer does with VNF instances what the internal load balancer does with the VNFC instances. From outside, the VNF is always seen as a single entity, but the VNF is replicated, and the load balancer is responsible for balance traffic among them. State considerations done

[10] https://www.tmforum.org/oda/business/process-framework-etom/ [Accessed: Nov. 08, 2022]

[11] https://www.tmforum.org [Accessed: Nov. 08, 2022]

before are valid also here. A completely different model is the end-to-end load balancer. In this case, N VNF instances are seen as N logical entities from a peer NF. This peer NF itself contains load balancing functionality, to scatter between these different logical interfaces. The Network Function Virtualization Orchestrator (NFVO), that will be discussed in the Section 2.3, can instantiate multiple VNFs, but it not responsible nor able to instantiate the load balancer. State considerations are still valid also in this case. Lastly, the infrastructure network load balancer provides that different VNF instances are seen as one logical entity from a peer NF point of view, but the load balancing is performed by a load balancer provided by the infrastructure. In this case, if the VNF and the NFVI support this mode, the NFVO can instantiate multiple VNFs and configure a load balancer in the NFVI that manage the traffic scattering for that group. In this case, as for external and end-to-end load balancer model, VNF replicas can be from different vendors. This is mainly done to increase resilience properties. This clause is not applicable for the internal load balancer model that, by definition of VNFCs, is a single VNF provider solution.

As aforementioned, it is due to cite also scaling-related aspects. According to ETSI, three models of VNF scaling are identified: auto scaling, on-demand scaling and scaling based on management request. Auto scaling provides that the scaling of the VNF is triggered by the VNF manager, according to the rules defined in the VNFD. An example could be a scaling triggered by rules based on resource usage monitoring. On-demand scaling provides instead that the VNF itself or its EM, monitoring the states of the VNFCs, raise specific requests to the VNF manager for scaling, that means increase or decrease the number of VNFC instances. Lastly, scaling based on management requests provides manually triggered scaling, for example a scaling triggered by a Network Operations Center (NOC) operator, via an appropriate interface. Both scale-in and scale-out operations are supported for all of the presented models [14].

2.3 Open Source MANO (OSM)

Section 2.1 and Section 2.2 introduced NFV concept, detailing the main features of single VNFs, introducing the infrastructure needed to deploy VNFs and presenting the improvements that these technologies can bring. Undoubtedly, each time that an improvement is introduced, it comes with brand new needs and requirements along with him, following the “nothing comes for free” principle of the engineering. In this scenario, since concepts as VNFs did not exist before the NFV coming, their handling and management requires a new and different set of functions and technologies. In this regard, ETSI defined a framework, namely the Network Functions Virtualization Management and Orchestration (NFV-MANO) and started an initiative named Open Source MANO (OSM) in order to develop this framework [15].

The NFV-MANO has the responsibility to manage the NFVI resources allocation for the various VNFs and therefore it plays an indispensable role because of the decoupling of the NFs from the hardware infrastructure and components they run upon. Resources allocation for VNFs could be a complex task because of the number of needs and requirements that VNFs might have at the same time. As an example, a VNF can need a low latency or high bandwidth environment to work properly, resulting in the introduction of constraints in the resources allocation for it, especially for network resources. Moreover, orchestration is a dynamic process, since the resources assignment to a particular VNF instance can vary during the lifecycle of that specific VNF instance, in parallel and consequently to an increase or decrease in the need for resources by that instance. As an example, a virtual NAT could need a additional memory resources during its lifetime, consequently to a raise in the number of hosts it has to masquerade, in order to maintain the desired performance and full functionality.

If from the NFVI point of view the orchestration and management aspects are related to the allocation of resources for the various VNFs, from the VNF perspective, the focus is on creation and lifecycle management of the needed virtualized resources and the VNF itself, altogether referred as VNF management. In the VNF management context, major operations are:

- VNF instantiation: the creation of the VNF and the assignment of initial needed resources to it;

- VNF scaling: the operation of increasing or reducing the capacity of the VNF (vertical scaling) or of increasing or reducing the number of running replicas of a given VNF (horizontal scaling);
- VNF update and/or upgrade: the operation of performing changes in the VNF software and/or configuration;
- VNF termination: the set of operations to be performed at the end of the VNF lifecycle, when the VNF-associated NFVI resources need to be returned to the infrastructure resource pool.

The deployment and operational behavior requirements for each VNF is defined in a deployment template, already introduced in the Section 2.2 and named VNFD, which describes the attributes and requirements that are necessary to instantiate and manage the lifecycle of the VNF. The VNF management functions perform the lifecycle management of the VNF based on the requirements specified in the deployment template. During the instantiation phase, the NFVI resources are allocated to the VNF according to what is declared in the deployment template and taking into consideration case-specific requirements or constraints that are pre-provisioned or are introduced by the specific instantiation request. The deployment template allows the VNF management functions to handle in a similar automated way very simple mono-component VNF or complex ones with multiple components and inter-dependencies. VNF Manager is also able to monitor Key Performance Indicators (KPIs) if these are specified in the template. Through the monitoring of one or more KPIs, the VNF management functions are able to understand the state of the VNF instance from the resources usage point of view and proceed with automated scaling operations. It is worth remarking that all the statements cited for VNFs are valid in general for network services. Network service orchestration is the set of operations that allow the creation, lifecycle management and disruption of a network service through the management of all its composing NFs, whether they are pure or virtualized.

Other important orchestration and management aspects of VNF include: fault and performance management, policy management and testing aspects. All of these functionalities need to be present in a NFV framework and, even if, in principle, they do not play a crucial role in VNF lifecycle, their presence in the ecosystem is essential because, respectively, they support the assurance aspects of the lifecycle of a VNF, give the possibility to introduce management rules of type condition-action to ease the whole management process through automation, allow problems detection in network functions and services before they are deployed in production.

Before going deeply in the NFV-MANO architectural framework description, the concepts of administrative domain needs to be introduced. In a usual scenario, the whole NFV system is not controlled by a single organization. An administrative domain can be mapped to different organizations and so it can be distributed among several service providers. Mainly, two major types of administrative domains exist:

- Infrastructure Domain: an Infrastructure Domain may provide infrastructure to a single or multiple Tenant Domains. The Infrastructure Domain is application agnostic and thus has no visibility of what is executed within a VNF.
- Tenant Domain: a Tenant Domain may use infrastructure in a single or multiple Infrastructure Domains. The Tenant Domain is application aware, in the sense that functional blocks in this Administrative Domain understand and support the logical functionality of the VNFs.

The identification of different Administrative Domains leads to the requirement to use different management and orchestration functionality in those domains. The cross-relationships between the different Administrative Domains leads to the identification of the requirement that NFV-MANO functionality is not monolithic, but rather needs to be layered.

After introducing NFV orchestration and management, and administrative domains concept, the NFV-MANO architectural framework needs to be presented deeply. This framework relies on a set of principles that support the combination of aspects of distinct administrative domains

and layered management/orchestration in each one of those domains. The architecture provides the possibility of introducing multiple NFV-MANO functional blocks in the tenant domain, each one leveraging resources in the same infrastructure domain, following a principle of equality, with no functional block that has primacy over others. The NFV-MANO functional blocks in the infrastructure domain should offer abstracted services to functional blocks in the tenant domain, embedding aspects of resources management as selectable resources policy within them. The functionality of NFV-MANO may be realized in different ways, from monolithic single instance to completely distributed implementation, or a scalable system with multiple load-sharing and load-balanced instances. It may be introduced as an extension of a cloud/network management system or as a stand alone system interacting with the surrounding cloud environment for NFV realization.

Figure 2.4 represents the NFV-MANO architectural framework. This representation given by ETSI is functional level, and does not imply any specific implementation. As an example, more functional blocks could be aggregated, internalizing the reference points between them, or some of the functional blocks can be replicated for quality of service reasons. Each one of the functional blocks being part of the NFV-MANO architectural framework has a well-defined set of duties and a specific role for providing the whole management and orchestration functionality and it with the other blocks offering them specific sub-functionalities or leveraging functionalities offered by the others.

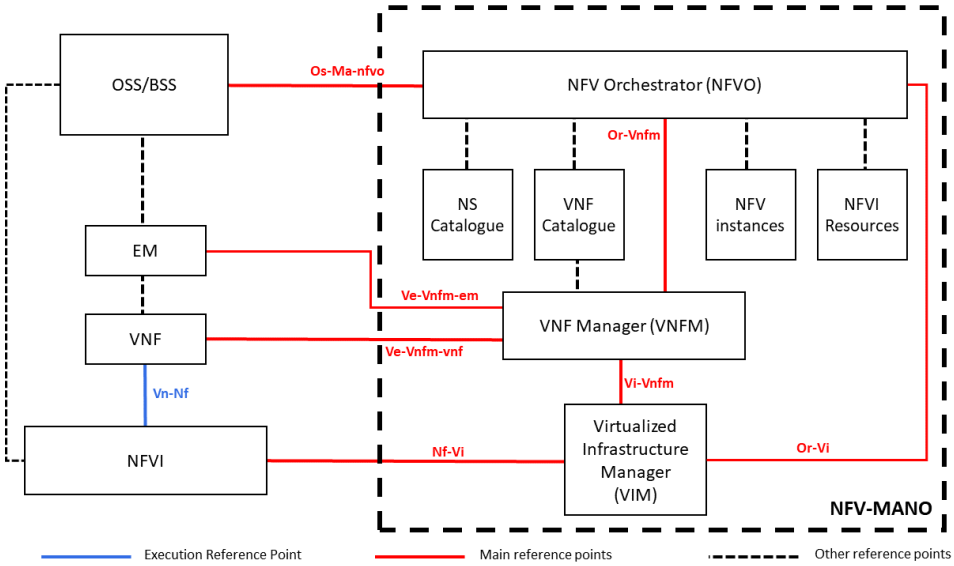

Figure 2.4. NFV-MANO architectural framework

As shown in the Figure 2.4, the internal structure of the NFV-MANO architectural framework consists of three components: NFV Orchestrator (NFVO), Virtualized Infrastructure Manager (VIM) and VNF Manager (VNFM). These components constitute the core of the NFV-MANO framework.

The NFVO is the beating heart of the NFV-MANO framework and it has two main responsibilities: orchestration of NFVI resources and lifecycle management of NS. The NFVO, through its resources orchestration functionality, provides services that support accessing the NFVI resources in an abstracted way, independently from any VIMs. In particular, the resource orchestration function of the NFVO offers the following functionalities:

- validation and authorization of NFVI resources requests from the VNF Manager;
- NFVI resource management, including distribution, reservation and allocation of the NFVI resources to NS and VNF instances;
- supporting the management of the relationship between the VNF instances and the corresponding allocated NFVI resources;

- policy management and enforcement for NS and VNF instances;
- collecting usage information of NFVI resources used by VNF instances.

On the other hand, the other type of NFVO functions, namely the network service orchestration functions, are responsible for all the needed actions for the lifecycle management of NSs. The main functionality offered by this group of functions is the management of network services deployment templates, as well as VNF packages, i.e. on-boarding new NSs and VNF packages. When a new NS or VNF is deployed, first, there is the need of a validation step: the procedure needs to check the integrity and authenticity of the provided deployment template and the consistency of all the information inside it. Moreover, this group of functions take care of the software image cataloging of the different VNF components inside the VNF package. Besides this main functionality, the network service orchestration functions of the NFVO provide the following capabilities:

- NS instantiation and NS instance lifecycle management;
- VNF Manager instantiation management;
- VNFs instantiation management;
- validation and authorization of NFVI resource requests from VNF Managers;
- management of the integrity and visibility of the Network Service instances through their lifecycle;
- NS instances automation management;
- policy management and evaluation for the NS instances and VNF instances.

As mentioned above, another core component of NFV-MANO is the VNFM. The main responsibility of this component is the lifecycle management of VNF instances. Each instance is intended to have an associated VNFM and the same VNFM may be responsible for the management of one or more VNF instances, even of different types. The overwhelming majority of the VNFM functions are assumed to be generic and applicable to any type of VNF. However, NFV-MANO has also the duty to support particular cases in which VNF instances need specific functionalities for their lifecycle management, and such additional and/or particular requirements may be specified in the VNF package. The deployment and operational behavior of each VNF is specified in the VNFD and there is a one-to-one correspondence between VNFD and VNF package. The VNFD describes the attributes and specific requirements necessary to deploy the specific VNF it is related to and also NFVI resources assignment to the VNF instance is based on the requirements specified in the VNFD, in addition to particular requirements, constrains and policies that are pre-defined or that come with the instantiation request (e.g. geo-location placement). It is worth remarking that, to guarantee the portability of VNF instances, hardware resources need to be properly virtualized and abstracted and also the requirements need to be described in abstract terms. Therefore, the VNFM is responsible for VNF instantiation, including configuration and feasibility checks if is required, VNF instance software update/upgrade, modification, scaling, collection of performance measurements and termination, besides to lifecycle management change notification. Lastly, the VNFM is also responsible for overall coordination for configuration and event reporting between the VIM and the EM associated to various VNFs.

The last NFV-MANO core component is the VIM. A VIM is responsible for computing, storage and network resource management, usually within one infrastructure domain, and it can be specialized for a single type of resources (e.g. networking-only VIM) or may be able to manage multiple types of NFVI resources. The VIM implementation is not specified by NFV-MANO, but the interfaces it exposes and the functionalities it must offer are in-scope. VIM main role in the whole ecosystem is to be the supervisor of the association between virtualized resources to the physical compute, storage and network resources and it keeps an actual inventory of all the allocations of virtual resources to physical ones. Moreover, it is responsible for managing of the virtualized resource capacity, intended as density of virtualized resources to physical ones, managing the software images as demanded by the other NFV-MANO functional blocks, performance and fault information about hardware, software and virtualized resources collecting and

reporting, and managing of catalogues of virtualized resources that can be consumed from the NFVI.

Once that the core components of NFV-MANO have been presented, it is due to introduce the concept of catalogues. In the NFV-MANO context, catalogues are a sort of repositories and each catalogue is responsible for keeping information about different types of data. As an example, the VNF catalogue role is to keep data about all of the known VNF packages and stores information such as VNFDs, software images or manifest files. In the same way, the NS catalogue stores information about NSs and can be queried by the NFVO to obtain NS or virtual link deployment templates, i.e. Network Service Descriptor (NSD) or Virtual Link Descriptor (VLD). NFV instances catalogue is the repository that keeps track of all the information related to VNF and NS instances. It holds two different type of records, related to VNF instance and NS instance and both of these type of records are updated during the lifecycle of the VNF or NS respective instances. Finally, the NFVI catalogue is responsible for holding information about NFVI resources availability, reservation and allocation, playing an important role in supporting the NFVO resources orchestration functions, tracking the NFVI resources allocation to associated NFV or NS instances.

Lastly, it is due to give an overview of the NFV-MANO internal and external interfaces, namely reference points, following the ETSI nomenclature. As cited in the Section 2.2, when presenting the internal architecture of a single VNF, also in NFV-MANO context the reference points are the point of contact between different components. On of the most important interfaces is the Os-Ma-nfvo reference point, which is the point of contact between the NFVO and the external world. Thanks to this interface the NFVO is able to receive, among the others, requests and commands about NSs and VNFs lifecycle management, forwarding the latter to the correspondent VNF Manager, through the Or-Vnfm reference point. In the end, it is worth citing the Ve-Vnfm-em and Ve-Vnfm-vnf interfaces, which are the point of contact in which the communications of (especially live) information between the single EMs/VNFs and VNFM [15].

Chapter 3

Lightweight Virtualization

In this chapter virtualization will be explored. In particular, the two solutions for the creation of virtualized software products will be presented: virtual machines and containers. First, a brief introduction about virtualization in general is given. Then, virtual machines functioning and insights are going to be presented, discussing how it is possible to implement them and retracing the history of the most widely known way to build virtualized systems. Subsequently, containers are presented as a form of lightweight virtualization, with particular focus on standard platforms in this context. For what is related to virtual machines, the x86 CPU architecture will be the reference, since it gives the possibility to discuss step by step the virtual machines technology, while for containers-related parts, Linux will be the reference operating system for this chapter, as presenting the solutions originally proposed by Linux can help in explaining and understanding modern virtualization platforms. Lastly, the final section will present several aspects specifically related to vNSFs, illustrating which of the two solutions is most suitable for vNSFs and the advantages and disadvantages introduced each one of them. It is worth remarking that, in this scenario, a vNSF is an application like all the other, so the following chapters use a common nomenclature using the most general term "application" when referring to a software product that runs in a virtualized environment.

3.1 Overview

As aforementioned in Chapter 2, cloud and virtualization technologies coming brought a new set of advantages in terms of flexibility, scalability, isolation and resource utilization optimization. After have introduced NFV concept and virtualization in general, it is necessary to explain how it is possible to implement in practice all the previously mentioned concepts to gain these benefits. Standards and frameworks presented in Chapter 2 do not give indication on how to actually implement the new virtualization technologies. One of the most common and most known solution when dealing with virtualization is to deploy all what is needed in the form of a Virtual Machine (VM). A VM is a software implementation of a real machine that can execute unmodified operating systems and applications as they run on a real computer [16]. However, another solution exists and it is particularly useful to overcome the limitations (especially in terms of performance and ease of deployment) that are introduced using the VMs. This alternative is actually represented by the operating system level, actually Linux, containers. A Linux container is a set of one or more processes that are isolated from the rest of the system. All the files necessary to run them are provided from a distinct image, meaning Linux containers are portable and consistent as they move from development, to testing, and finally to production[1]. Thanks to the container technology it is possible to introduce a form of lightweight virtualization.

First, it is necessary to introduce that the virtualization in general born because of the need of isolation between multiple applications and the optimization of resource utilization. Before the

[1] https://www.redhat.com/en/topics/containers/whats-a-linux-container [Accessed: Mar 21, 2023]

virtualization wide-spreading, the most common solution followed by providers was the so called "one application per server rule". This practical rule imposed to deploy a single software application for each physical server, to ease the application management, but especially it derived from the fact that software was, and sometimes still is also nowadays, tested for a specific environment (e.g. operating system above all).

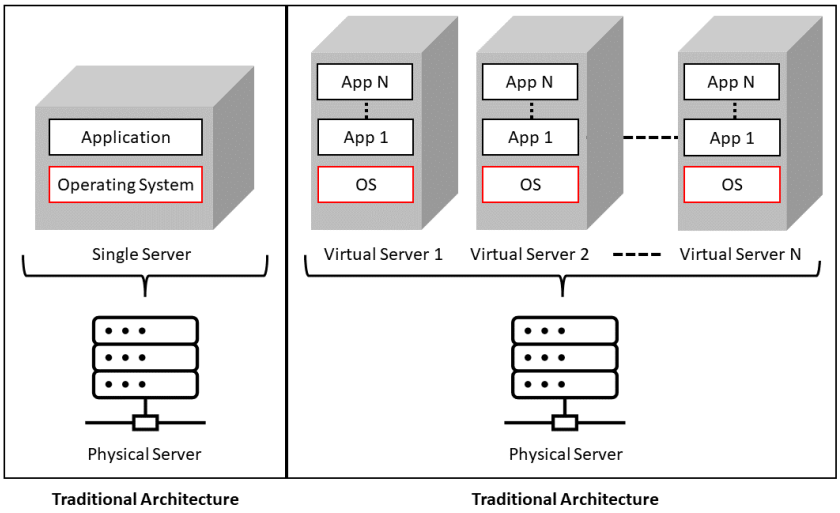

Figure 3.1. Traditional one application per server architecture and new virtualized server architecture

This situation was limiting, and started to become a problem especially for expanding companies with an IT infrastructure that was beginning to become important in terms of number of hardware. In addition, it is worth remarking that at that time the overall IT infrastructure organization had not an accurate organization and, in most cases, servers were installed across the whole company. The first solution adopted was the so called "consolidation". Trivially, the first idea was to consolidate as much hardware as possible in a single physical location, even a single room, to ease the management of the servers and optimize physical space and appliances like network wires.

In this scenario, with the vast majority of the hardware that started to be consolidated in single locations, providers realized that there was so much hardware around and it was mostly idle. This idleness was mainly due to the aforementioned "one application per server" rule, born as a solution to the popular Operating Systems (OSs) failure in providing appropriate configuration, temporal and spacial isolation. In fact, the processes schema given by the OSs does not meet the appropriate requirements for isolation of applications, since the performance of a process affect the performance of another and if a system failure or a simple crash happens to a process it may compromise another process. Moreover, there is also a question related to system requirements, e.g. one application needs a specific library version and another application needs a different version of the same library: one of the two applications will not run in the environment it was tested in and the vendor will not be responsible anymore for malfunctioning.

Finally, this scenario introduced also another problem: at a certain point of the technology evolution (late 90's), the CPU industry started to adopt the multi-core technology, since it was impossible to significantly further increase the CPU speed increasing the clock frequency [17]. However, in this context, the vast majority of the applications were still single-threaded and not able to exploit the simultaneous presence of several cores. The result was a huge amount of servers, for the overwhelming majority underutilized (due to impossibility of application to exploit the multi-thread technology and the one application per server rule), and consuming a lot of electrical power. Hence, the overall problem was the impossibility to consolidate multiple apps on the same hardware server. The main problem in this situation was definitely due to the power consumption: it has been demonstrated, also using different server configurations, that the power

consumption for an idle server's CPU is often near 2/3 and generally never below the 50% of the power consumption of the same server's fully loaded CPU: i.e. it is better to maximize the CPU load [18]. The success of virtualization depended on this: it was the perfectly suitable solution for this set of problems.

3.2 Virtual Machines

After a brief introduction about virtualization and after having introduced VMs and containers, it is necessary to go deeply in the presentation of the first of these two technologies and to present the advantages and disadvantages brought by each one of them.

As above mentioned, a VM is the most traditional, but not the only one, way to deploy a virtualized software product. VMs usage was popular in the early 60s [19], because it allowed to efficiently share the resources of big mainframe computers and run multiple operating systems (at the time, single user) upon a single physical system. The interest in this technology was lost with the appearance of the modern multi-user operating systems and the drop in the hardware costs, which brought to the diffusion of personal computers and the gradually loss of hardware support for virtualization.

VMs run on top of a virtualization software layer named hypervisor. The glossary made by Vmware[2], one of the biggest companies which deal with virtualization technologies and develop virtualization-related software, classifies the hypervisor as a "virtual machine monitor" (VMM). The hypervisor is a software process that is responsible for the creation, execution and management of one or more VMs and it is the software layer that makes it possible to host multiple VMs on a single host computer. Two types of hypervisors exist: so called "bare-metal" hypervisors run directly on the host hardware, "in hosting" hypervisors run instead as a software component of an operating system, such as any traditional process[3]. In this scenario, the physical system on which an hypervisor is installed and which hosts one ore more VMs is called host, while the hosted VMs are called guests. The hypervisor manages the system resources (i.e. computing, memory and storage resources) as a virtual resources pool it can allocate to the guest VMs, even dynamically, allowing a host computer to support more than one VM through virtual resources sharing (i.e. a level of abstraction of the hardware resources is introduced). The abstraction of hardware resources is what optimizes the resources consumption within a system, since this makes it possible to assign to VMs more resources than the actual present in the system. The virtual resources sharing also provides benefits such as improved applications portability, since physical characteristics of the system are hidden from the applications, which only perceives the virtualized resources.

The hypervisor job, and consequently the VMs existence is based on the physical CPU, memory and storage (mainly, in general all I/O resources are included) resource sharing and virtual CPU, memory and storage resources assignment to each VM. Another constraint for the hypervisor is given by the "full virtualization" paradigm. This virtualization technique makes it possible to completely decouple the software from the hardware it runs on and, if this paradigm is followed, the guest OS should not be aware of not running on a physical machine but on a virtual one.

3.2.1 CPU virtualization

The VMM needs to assign one or more CPU core to the VM to be able to execute the guest operating system. At this point, for the record, it is mentioned that the Instruction Set Architecture (ISA) of the virtual CPU could also be different from the physical one and this result could be achieved relying on "emulation" and not on virtualization. The emulation of a different ISA requires binary translation, i.e. the translation of the binary instructions of one ISA into another, to make it understandable and executable for the host CPU.

[2] https://www.vmware.com/it/topics/glossary/content/hypervisor.html [Accessed: Mar 30, 2023]

[3] https://www.vmware.com/topics/glossary/content/bare-metal-hypervisor.html [Accessed: Mar 30, 2023]

Taking the x86 Intel CPU architecture as a reference, the CPU virtualization is performed by the hypervisor in several ways. x86 CPU architecture provides four level (called rings) of privilege: the ring 0 is the privilege level where the OS kernel runs, the other rings are designed to run applications with decreasing level of privilege [20]. Actually, modern OSs make use of ring 0 and 3 only.

The main question when dealing with CPU virtualization is related to the privileged instructions: instructions that need to be executed in a privileged hardware context, i.e. the HALT Intel CPU instruction, used to halt the system, cannot be run by a user process. A privileged instruction needs to be executed within a privileged hardware context, otherwise a trap is generated. This trap is launched by the CPU to notify the surrounding system that an instruction has been executed in the wrong hardware context. The host OS (or the VMM) could intercept and interpret this trap, in order to handle it and to run the trapped instruction at a "lower" ring. This implies that a guest OS running at a ring greater than 0 cannot directly handle and execute privileged instructions. Virtualization makes use of ring de-privileging technique, which provides that all guest software is run in a ring greater than 0, to avoid conflicts between guest software and VMM. Hence, a guest OS run inside a VM cannot handle privileged instructions.

The traditional paradigm invented for virtualization is the Trap & Emulate (T&E) [21]. The guest OS is executed in an unprivileged hardware context and when a privileged instruction execution is needed, a trap is generated by the CPU. Subsequently, the trap is intercepted and handled by the VMM, which runs at ring 0 and emulates the privileges instruction, and then give back the control to the guest OS. Moreover, the VMM must handle also system calls triggered by user applications running upon the guest OS, since the latter is not able to handle them, due to the fact that it runs at a greater than 0 ring. The system calls in question are intercepted by the host OS kernel, which emulates the system call execution and then gives the control back to the guest OS, which suddenly returns the result to the guest user application. Privileged instruction handling is the major source of overhead when using VMs and the execution of a system call coming from an application running on the guest OS could require until 20 times the time to execute a traditional system call [22].

Apart from the overhead, another question is introduced by the presence of sensitive instructions: instructions that leak information about the physical state of the CPU, without generating a trap. In this situation, a VM could access, without the VMM supervision, information about physical hardware and potentially make decisions based on this information, leading to errors (i.e. an application running on a guest OS detects 16 CPU cores based on information retrieved through a sensitive instruction, than it decides to launch 16 thread, but only 2 cores are assigned to that machine and at the end the application executes slower due to continuous thread switching).

These considerations about overhead introduced by the privileged instructions and confusion introduced by the sensitive instructions leaded to declare the x86 architecture as not virtualizable. Essentially the problem resided in the fact that the VMM was not able to emulate correctly using the traditional T&E paradigm. Obviously, starting from these criticalities, several solutions have been studied and developed.

Among the various solutions that have been studied for this problem, the most interesting are Dynamic Binary Translation (DBT), paravirtualization and hardware-assisted virtualization [23].

DBT is the process of dynamically translating machine code (binaries) belonging to one ISA to another, for the purpose of running such machine code written for one ISA on a different one. DBT follows a full virtualization approach: the translation is performed on the fly and binary code level, so the guest OS does not need any modification to run in a virtualized environment. The systems performing DBT are usually referred as emulators, hence, it has been demonstrated that DBT is a more efficient form of emulation[24]. DBT approach guarantees compatibility since there is not the need for specific hardware support or modifications to the OS source code. On the other hand, despite to the improvements introduced compared to raw emulation, still introduces a significant overhead, always related to the system call execution.

A completely different approach is the paravirtualization: the process of strategically modifying a portion of the interface that the VMM exports along with the OS that executes using

it. This solution deeply simplifies the virtualization process, since the guest OS source code is modified and it is aware of when it is necessary to call the hypervisor, avoiding the significant overhead introduced by the system call dribbling between the guest OS and the VMM: i.e. system calls and non-virtualizable instructions are replaced by specific hypervisor calls. The cost of this high-performance solution is represented by the effort that is needed to modify the OS source code. Therefore, this solution implies that only OSs with modifiable source code can be paravirtualized (e.g. Windows cannot be virtualized with this solution) and suddenly it does not support some OS.

Nowadays, the most efficient and most solution is indeed the hardware-assisted virtualization, sometimes referred also as native virtualization or Hardware Virtual Machine (HVM). Modern HVM improvements introduced significant reduction of the overhead due to the system calls execution. Both Intel and AMD introduced support for virtualization on their CPU architectures. Essentially, HVM is implemented by the introduction of a new number of primitives in the ISA of the CPU to support the classical VMM, creating an in-memory data structure called VMCB (VM Control Block) that combines the control state of the physical CPU with a subset of the state of the virtual CPU and the creation of a new, less privileged, execution mode called guest mode (VMX non-root in x86 architecture) in addition to the traditional execution mode (VMX root in x86): i.e. referring to the x86 architecture, a new couple of instructions (VMentry, VMexit) transfers the CPU execution mode from host to guest and vice versa[25]. The privilege rings structure of the x86 CPU is duplicated and is organized as follows: the guest OS user applications run on ring 3 and the guest OS runs on ring 0 in VMX-non root mode, while the VMM runs on ring 0 in VMX root mode. In this way, now the guest OS runs at ring 0 and is able to intercept and handle the system calls coming from the user applications running upon it. The hardware is always managed by the VMM, which only takes action when the guest OS calls it, due to the impossibility of satisfying a system call. Summarizing, HVM introduces a simple and transparent way to implement the full virtualization approach and minimizes the VMM intervention, while the round-trip time for VMentry/VMexit operations represent the major source of overhead for this solution. CPU manufacturers are investing many resources nowadays for the implementation of always more efficient VMentry/VMexit operations, to reduce the cost in term of performance. HVM is indeed the most cutting-edge solution for virtualization nowadays.

3.2.2 Memory virtualization

Apart from the CPU virtualization, explained in Subsection 3.2.1, which represents the major point of discussion when dealing with VMs, it is due to mention also aspects related to memory and I/O virtualization. As well as the computing facilities provided by the CPU, a VM also need memory and I/O facilities to be fully up and running. For memory-related facilities, the transition to virtualization was much more immediate than it was for CPU aspects. In fact, memory virtualization was introduced even before the coming of new requirements due to VMs. Memory virtualization is a technique to access as contigous dispersed memory locations in the physical memory and it was introduced to make a process perceive more (virtual) memory than the actual physical memory available for it. Many CPU architecture, as well as the reference for this paper x86, are able to handle this task directly in hardware thanks to a dedicated Memory Management Unit (MMU). When the CPU needs to access a memory location, it receives a virtual memory address from a process on the address bus. This virtual memory address is then split as shown in Figure 3.2 and each group of bits of the virtual memory address bit string is used as an offset to access specific location of a table in cascade and the CPU, which stores the physical address of the top level page table in one of its registers, is able to resolve the actual physical memory location.

Thanks to the presence of this technique which was already available for standard process, to deal with the transition to virtualization it is sufficient to introduce an additional level of translation: i.e. the guest OS user process virtual address is resolved to a guest OS physical address, which is then turned into a machine physical address, always using the same technique in an iterative way. To avoid this double address translation, another data structure can be introduced: the shadow page table. This additional table maps guest virtual addresses and machine physical pages. The world “shadow” provides an idea on how this table is used, in fact,

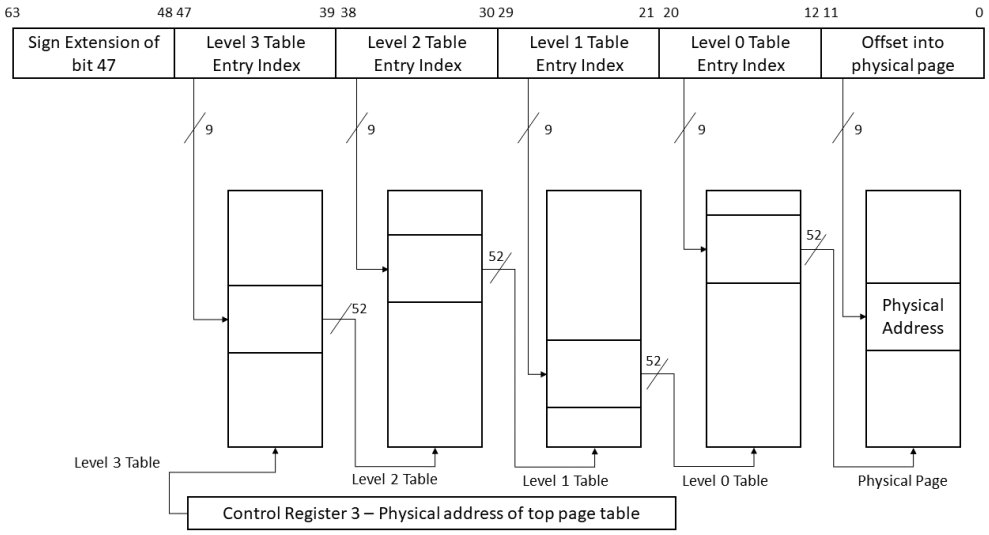

Figure 3.2. x64 CPU architecture memory paging

it is a page table that is not visible from the guest OS point of view and is only used by the CPU when a translation from a guest OS is needed. The shadow page table is managed by the VMM, which is in charge of synchronize it with the guest OS page table, whilst the host OS will not modify its ordinary page tables. This solution, however, introduces a significant overhead in presence of page faults, that is when a required memory page is not actually stored in the physical memory at the moment in which it is asked to access it. In these cases, the memory page needs to be loaded from the disk drive. The overhead is due to the fact that and extra page table needs to be kept synchronized when a page fault occurs, which leads to an additional VMexit. For this reasons, new technologies as EPT[4]/NPT[5] (Extended Page Table/Nested Page Table) have been introduced. With these solutions, the traditional page tables are able to translate directly from guest virtual address to physical memory address, avoiding additional VMentry/VMexit trips when the page table needs to be updated, since here the guest OS is able to update its own page table without the VMM intervention. The cost for the performance improvements introduced by EPT/NPT is the need of a double-level page walk: multiple memory accesses are needed when an address needs to be resolved and "hidden" memory access happen each time an address translation is needed. In this context, the efficiency of the Translation Lookaside Buffer (TLB), measured with the "hit ratio" becomes crucial. The TLB is a sort of cache for the page table, which maintains the direct mapping between a virtual page memory location and the physical correspondent one, based on the last translations performed and speeding up the memory access process. However, memory-related aspects are usually not critical since a modern TLB is designed to be small enough so that a linear search through it takes less than one CPU cycle and is able to reach hit ratio even greater than 99% [26], thanks to the exploitation of principles as the locality of reference.

3.2.3 I/O virtualization

The last aspect to take into account when talking about VMs is the I/O virtualization, in which also the virtual storage aspects fall. Several techniques have been studied to solve the problem of the virtualization of an I/O device. In particular, the three most important solutions can be

[4] https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html [Accessed: Mar 19, 2023]

[5] https://www.amd.com/system/files/TechDocs/33610.PDF [Accessed: Mar 19, 2023]

identified in the following: device emulation, para-virtualized device and direct assignment [27]. The choice of a particular solution depends on various aspects as the type of the device and if the device is shared between different OS or dedicated to a single one.

Device emulation, as well as the emulation process introduced when dealing with the CPU virtualization, follows a fully virtualized approach: the guest OS is not aware it is using an emulated device and it will use the same device driver used with the correspondent physical device, then the VMM is responsible for the translation and emulation process remapping the device communication with the physical device. Similarly to CPU emulation, this solution is easy to implement and to set up, since there is no need for additional dedicated drivers and a single device can be multiplexed and be associated to multiple virtual devices. However, this solution introduces a significant overhead in terms of I/O execution time, increasing the CPU load, since the processor is responsible for the emulation. A completely different approach is the para-virtualized device one. This solution provides that the guest OS is enriched with dedicated drivers and follows an approach similar to para-virtualization: i.e. the additional drivers are composed by a frontend (installed in the guest OS) and a backend (installed in the hypervisor) and these two components of the driver communicates using a protocol optimized for virtualized environments. Finally, the direct assignment solution allows the VM to directly communicate with the physical device. The guest OS is fully responsible for the device handling using the traditional drivers and the overhead introduced by the virtualization is actually null. Obviously, this solution requires that the physical hardware device is exclusively assigned to the specific VM, which is indeed a strong limitation. Summarizing, the direct assignment solution is indeed the one which guarantees the highest performance, but it needs the use of dedicated devices. For this reason, the PCI-SIG introduced a new standard in 2016, named SR-IOV[6] (Single Root I/O Virtualization and sharing). This standard provides mechanisms to create natively shared devices, handling the device multiplexing directly in hardware.

3.3 Linux Containers, Docker and Kubernetes

As cited in the Section 3.1 of this chapter, containers represent the alternative to virtual machines when dealing with virtualization. Containers are based on a completely different paradigm: the lightweight virtualization. This technology allows to isolate processes and share resources without the intrinsic complexity of virtual machines. In 2008 Linux, that will be taken as a reference OS here, introduced LXC (LinuX Containers)[7]. LXC is an OS-level virtualization method that allows to run multiple isolated Linux systems (precisely, containers) on top of the same Linux host providing an isolated operating system environment with its own file system, networks, process tree and I/O space. A container task is to group processes together inside an isolated environment which includes all what is needed for this processes to run (executable files, libraries, ...).

In this context, for security reasons, the achievement of strong process isolation turns out to be crucial. An example is the necessity of avoiding the interference between processes and services in case of failure or intrusion in one of them or of running untrusted/unknown programs on the system. Scenarios like these require strong process isolation, without the necessity to adopt heavy techniques like the full virtualization: i.e. there is no need for a different operating system. To address these requirements, two features are offered natively by the Linux kernel: control groups (cgroups) and namespaces.

Citing the Linux kernel documentation[8] "cgroup is a mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner". The main feature cgroup offers is the resource limiting for a single or a group of processes: specific group of processes can be set to not exceed a well defined memory usage limit,

[6] https://www.intel.com/content/dam/doc/white-paper/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.pdf [Accessed: Mar 28, 2023]

[7] https://www.redhat.com/en/topics/containers/whats-a-linux-container [Accessed: Mar 25, 2023]

[8] https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html [Accessed: Mar 31, 2023]

CPU load limit or network usage. This is particularly useful for resource sharing control, but also for other aspects as security. As an example, when running an unknown program, assigning it to a cgroup which limits its memory resources usage to a well defined given amount of bytes can protect from DoS-like attacks as forkbomb. Cgroups offer also the possibility to measure and monitor the resources usage by a group of processes, which could be useful for billing purposes. Prioritization is also introduced since cgroup gives the possibility to assign different CPU quotas, I/O throughput or network bandwidth to specific group of processes based on their importance, and control features are provided since through cgroups is possible to handle freezing and restarting of a whole group of processes.

Namespaces are a feature of the Linux kernel, highly relate to cgroups, which separate processes so that they cannot perceive resources of a given class (depending on the given namespace) in the other groups. With respect to cgroups, namespaces feature enables the creation of distinct virtual environments based on what is isolated for a group of processes with respect to other groups. Two different namespaces could have completely different view of the file system, or completely different network stack or completely different view of the process tree. Currently, 7 types of namespaces are provided by the Linux kernel:

- process namespace: in Linux there is a single system-wide process tree, each process has a parent until 1 (init process). Process namespace, also referenced as PID namespace, enable multiple nested process tree, where each process tree represents a completely isolated set of processes, hiding the entire process tree to processes belonging to a different PID namespaces;
- network namespace: allows processes to perceive a completely different network setup, so creating multiple network namespaces allows to completely separate group of processes from the network point of view;
- mount namespace: enables the creation of a completely different filesystem, starting from a copy of the original one;
- UTS namespace: is a very simple and minimalistic namespace which provides the isolation only of two system identifiers: hostname and NIS domain name;
- user namespace: allows a process to have root privileges within the namespace, without giving it that access to processes outside of the namespace;
- IPC namespace: gives the possibility to create a private inter-process communication resource exclusively for the process belonging to the namespace;
- cgroup namespace: provides a mechanism to virtualize the view of cgroup mounts and of the "/proc/$PID/cgroup" file. Without cgroup namespace, that mentioned file shows the complete path of the cgroup of a process, potentially leaking system level information which should not be visible when the purpose is to isolate the processes.

The idea behind lightweight virtualization is to create a system that can guarantee the same computer virtualization properties that have been presented for VMs (scalability, flexibility and isolation) but consuming less resources and introducing less overhead. This idea is essentially implemented by using operating system-level or application-level virtualization techniques instead of full machine virtualization approach. In the former case the hypervisor becomes the Linux kernel itself and there is not the need of an additional hypervisor. Lightweight virtualization actually comes with the same requirements as full virtualization: a fine-grained control of physical hardware resources of the host system and security and isolation guarantees. The difference is that when leaving full virtualization for the lightweight one, the virtualization is fully handled by the OS kernel and there is no more hypervisor that is responsible for these tasks. This means that all the features needed to implement virtualization must be present inside the kernel itself.

For this purpose, the aforementioned kernel features of cgroups and namespaces, originally developed to strengthen process isolation, can be leveraged. Combining the features offered by cgroups and namespaces is possible to control all the properties dictated by the requirements, such as partitioning of CPU and memory quotas or network/file system isolation. Despite this, the cited

solution of using raw cgroups and namespaces presents several limitations, since it is suitable only for single-host virtualization and cannot be used to manage more complex and distributed entities like datacenters, nor guarantee application portability. Moreover, the configuration is really hard (with many lines of code to be written) and not user-friendly. Cgroups and namespace, however, represent the basis to build lightweight virtualization, since they are able to offer all the needed kernel features with respect to isolation and resource management.

Leveraging kernel features offered by cgroups and namespaces, as above mentioned in this section, in 2008 Linux introduced LXC. LXC is in fact realized combining the cgroups and namespaces feature and implementing user-friendly configuration tools in user space, which give the possibility to use all of these features. Actually, all the necessary kernel level features for lightweight virtualization implementation were already present when LXC arrived, but LXC, even if it does not introduce any new feature, provides a way to group all the features in a user-friendly way. As an example, before LXC, cgroups was the the component to handle resource assignment, then namespaces should have been used to add isolation, other components as SELinux needed to be used to manage security and so on. LXC groups all the features offered by these components under a single user-friendly and user-space tool. However, LXC still shows some important limitations. Raw containers are not suitable for being migrated from a system to another, since there is not a feature for this in the Linux kernel, nor they have been designed to be portable. In addition, the resources isolation is not as precise as it was when using full virtualization and the amount of resources assigned to a given container could be affected by the behavior of others.

3.3.1 Docker

So far, LXC represents an interesting low-overhead alternative to VMs. However, this technology shows several limitations, especially related to application portability and migration and, most important, the configuration and management of containers present difficulties when the application starts growing. The solution to these problems, and the real reason for the vast widespread of containers is represented by the advent of Docker[9]. Docker is a platform designed to help developers build, share and run modern containerized applications. Docker focus is oriented to applications and its goal is to create fully lightweight, portable and self-contained application packages which run anywhere the Docker environment is present indistinctly, independently from the underlying OS or hardware configuration. The goals of this platform consist in building a unified environment to handle the applications lifecycle, provide adequate application isolation and transparent networking to the apps running inside containers. It has to be mentioned that Docker is not a virtualization engine, but it leverages the existing virtualization primitives, as cgroups and namespaces.

Before Docker, application packaging and distribution was not so straightforward. First, the large-scale deployment of an application required that this had to be packaged in several different formats to guarantee compatibility with each available OS. Moreover, for each OS, even different distributions had to be considered: e.g. on package for Debian, one for RedHat, and so on. Finally, there was the need to always create brand-new packages as soon as new OS version was released. In this context, dependencies represented the problem, since they may be different potentially for each distinct package and require libraries which are not present in all versions of OS/distribution/version. The strength of Docker is then represented by the portability it guarantees to applications, by making them self-contained and packaging them with all the dependencies they need, this is how the application is able to run anywhere indistinctly. To give an idea, also Docker slogan is “Develop faster. Run anywhere”.

As aforementioned, Docker leverages the virtualization primitives offered by the Linux kernel. Compared to raw LXC, Docker introduces an additional set of features, besides the cited application portability. In particular, it offers qualities such as automatic build, versioning, component re-use, sharing mechanisms, resource management and isolation, integration with major cloud vendors and compatibility/interoperability with products as Kubernetes[10], that is presented later in

[9] https://www.docker.com/ [Accessed: May 04, 2023]

[10] https://kubernetes.io/ [Accessed: Apr 16, 2023]

this section. An important concept when presenting Docker is the one of Docker image. A Docker image represents an immutable template for containers in the form [registry/][user/]name[:tag]. Starting from an image, containers can be actually run and their lifecycle management can be fully controlled via the Docker commands and a running container maintains changes within its own local file system. A container, in Docker context, is then a running instance of an image, which consists in the set of entry state specifications. Docker images can be maintained in a Docker registry: i.e. a space similar to a software repositories which stores Docker images. The main Docker registry is public and is named Docker Hub. An image can be built starting from a running container and then "capturing" its current state or by writing a Dockerfile. Dockerfile is a text file, formatted in a Docker-understandable way, that specify the instructions to automatically create a container. At this point, it is worth remarking that Docker containers share the same kernel as the host OS. The container must only "add" the subset of the OS it needs. In this way, images can be created in such a way that it has only the software that it strictly needs, avoiding the default software that is installed in a usual operating system.

Docker approach from the software architecture point of view consists in a client server architecture. As shown in Figure 3.3, the Docker client interfaces with the Docker deamon, which is the actual responsible for container deploying and lifecycle managing. It is due to cite that the client and the deamon could run on the same machine or it is also possible to connect a Docker client to a remote Docker deamon. The communications is based on REST APIs, over UNIX sockets in the former case or obviously network interface in the latter case. The Docker deamon, namelly dockerd, listens for Docker API requests and handles any kind of Docker object. In addition, a Docker deamon could also communicate with additional deamons in presence of Docker services. The Docker client is the access point for users to the Docker engine. Launching Docker commands such as the one to build an image (docker build), to run a container (docker run) or to create a new network (docker network create), the Docker client actually sends REST API requests to the Docker deamon, which understands the requests and performs the necessary actions. The Docker client can be also configured to communicate with more than one deamon. Alternatively, a Docker desktop application is available and it contains the client, the deamon and additional tools such as Docker Compose.

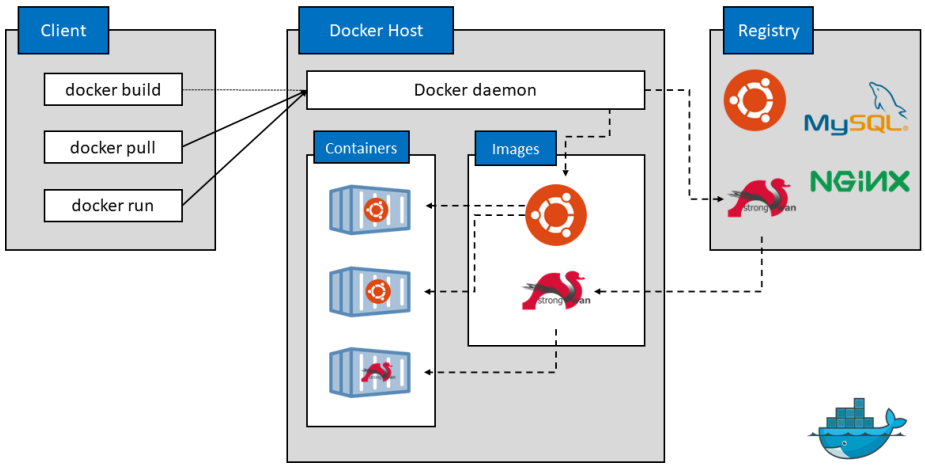

Figure 3.3. Docker software architecture

Docker network-related aspects are particularly interesting from the vNSFs point of view. By default, each running Docker container is attached to a default virtualized Linux bridge, named docker0. Docker automatically creates its own private networks inside the host, with its own address space, routing tables and firewalling rules (using iptables) to provide outbound internet

connectivity to containers, while inbound connectivity is automatically guaranteed. Network-related aspects are handled by the libnetwork component, which implements the Container Network Model (CNM): a standard container networking model proposed by Docker which offers several features such as:

- network sandbox: isolated environment which contains the network configuration of the container. It can be assumed as a networking stack inside the container;
- endpoint: a "pair of" network interface. It can be assumed as a virtualized Ethernet wire. It allows to connect one end of the wire in the network sandbox and the other one in a given network. This ensures that endpoints join only one network for each interface;
- network: a group of endpoints which are able to communicate with each other;
- user-defined labels: CNM also provides a labeling system. Labels are defined by the user and passed as metadata between drivers and libnetwork, to enable the runtime engine to inform driver behavior.

Docker networking permits to fully exploit the advantages given by the virtualization of network components. In fact, in Docker, each container can be attached to an arbitrary number of networks, making the deployment of complex VNFs, potentially presenting many network interfaces (e.g. for a firewall, input, output, management), more immediate since these interfaces can be configured by a few lines of code.

3.3.2 Kubernetes

As mentioned in Subsection 3.3.1, Docker permits to fully take advantage of the features offered by the LXC, providing full application portability, introducing a user-friendly platform which makes the management of the containers much easier and introducing further new features as the concept of automatic build. Docker has become the standard platform for building and management of container-based applications. However, when moving to modern large cloud environment, such as the datacenter of a TSP, Docker alone is not enough. This is because Docker is a single-host platform, while when moving to larger environment there is need for a distributed solution through which it is possible to handle the entire datacenter as a single entity. When talking about Docker above, the fact that it provides integration with other products, in particular Kubernetes, has been mentioned. Kubernetes (usually abbreviated as K8s) is a portable, extensible, open source platform, introduced by Google, for managing containerized workloads and services, that facilitates the automation.

Before going deeper inside K8s, it is due to introduce the concept of microservice. Back in the old days, applications were built as monolithic objects, which made the software development and testing processes slow and heavy, since the update of a single software component, or the introduction of a new one, implied the need for testing the whole application. Amazon, back in 1998, introduced the concept of microservice, publishing "The Distributed Computing Manifesto"[11]. This document introduces a new three-tier architecture, also called service-based architecture, where presentation, business logic and data are loosely coupled, to overcome the limitations of the two-tier client-server architecture. To have a client-server architecture implies that the application and the data model are really tightly coupled, which implies that each change in the latter provokes the need for a change also in the former, introducing limitations in terms of scalability. The new architecture proposed provides, instead, that the clients would no longer be tight to the data model directly, but only through well-defined interfaces that encapsulate the business logic necessary to perform the action. In this way, the client is no longer dependent on the underlying data. Hence, a change in the business logic to database interface does not affect anymore the client, or to introduce a change in the client, it is not necessary to modify the data model.

[11] https://www.allthingsdistributed.com/2022/11/amazon-1998-distributed-computing-manifesto.html [Accessed: May 18, 2023]

This document represents the groundwork for microservices and platforms as K8s. Microservices are the most widespread way to build modern applications. Following a microservice-oriented development approach consists in arrange an application as a set of loosely coupled services, abandoning the old monolithic approach and stepping into fully decoupled components, gaining improvements in terms of flexibility. Adopting a microservice-oriented architecture, each microservice is responsible for a single task and the communications between it and the others is performed using lightweight mechanisms such as REST API requests. Following a microservice-oriented approach introduces several advantages, under several points of view, as the simple separation of concerns, the improved scalability or the agile maintainability. The main advantages introduced by microservices with respect to monolithic approach can be identified in the following[12]:

- scalability: each microservice can be scaled by need independently of the others, while using monolithic approach the whole application needs to be scaled;
- resiliency: if a microservice fails, just the functionality offered by the specific microservice is broken;
- maintainability: microservices are almost totally independent each other, this ease the maintainability, while using monolithic approach the whole application code becomes difficult to maintain when it starts growing;
- deployment: each microservice is deployed independently, while with the monolithic application the whole app needs to be restarted every time each minor change is introduced;
- technological flexibility: each microservice can be developed using the most suitable programming languages and frameworks, while the monolithic application is usually written using a single programming language, introducing obvious limitations.

K8s represents the standard platform used nowadays to build microservice-oriented application. K8s follows a fully declarative approach, following the so called "infrastructure as code" philosophy, i.e. what has to be built is declared (the logic), not how it has to be built (the control flow). What is not explicitly specified by the client is defaulted by the platform using coherent values (defaulting). K8s follows a control loop-oriented approach, repeatedly checking the state of each component and checking the differences between the current and the desired state (enforcement), eventually performing actions to make the state converge (reaction). The control loop is implemented by K8s controllers, which take care of observing the current state of the whole environment (named cluster) interfacing with the API server, a component of the K8s platform which is responsible for responding to the API requests coming from other components, querying the knowledge base, named "etcd".

All K8s resources have the same structure and the type of object is uniquely identified by the "apiVersion" and "kind" attributes. In addition, each component has a "spec" and a "status" composed attributes and it is accompanied with two kinds of "metadata": "labels" which contain information relevant to users and "annotations" which are are used to enrich the basic specification.

The basic execution unit in K8s is the Pod. A Pod represents a small set (one or a few more) of highly coupled containers and disk volumes. Usually, a Pod consists of a single container, but since containers inside the same Pod share the IP address and localhost, the communication between them is facilitated, therefore for highly coupled containers, named sidecar containers (e.g. a web server and an authentication proxy) a single Pod with two or more containers is built. A Pod is always executed on a single node and represents for K8s what a process is for an OS. Furthermore, It is due to mention the importance of "init" containers: they are containers that always run to completion and sequentially each other in a Pod before the application containers run, usually in order to set up the environment, performing actions as installing specific utilities or tools the application needs to run.

[12] https://martinfowler.com/articles/microservices.html [Accessed: April 20, 2023]

K8s offers many built-in components, each one with its own specific behavior and functionalities, that can be used to manage certain application aspects. Each K8s component represents an abstraction and gives the possibility to adopt a declarative approach and abstract away some strictly container-related aspects. Usually an application in K8s is composed as specified in Figure 3.4. From a vNSF point of view, the most interesting components are ReplicaSet, Service,

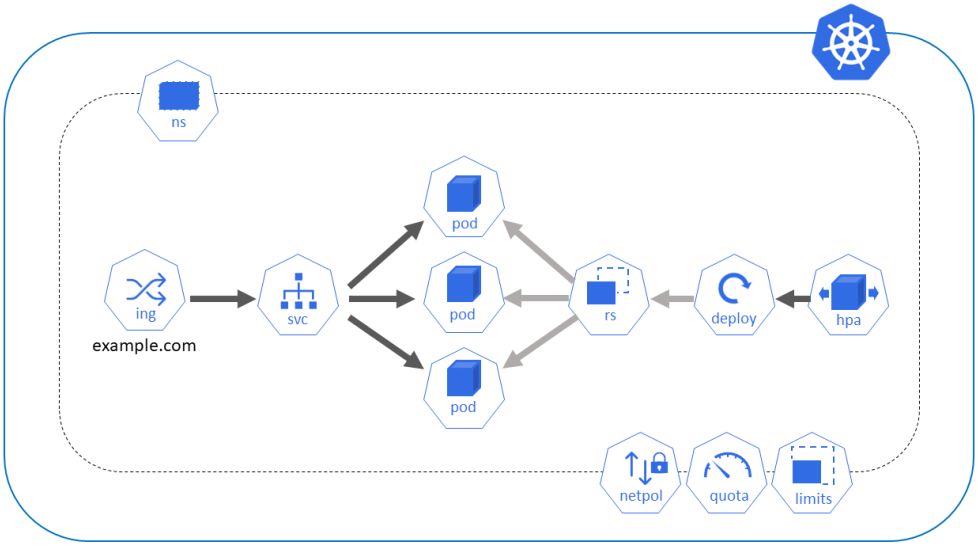

Figure 3.4. Traditional application structure in Kubernetes

Configmap and Secret. ReplicaSet is the component that allows to implement fault tolerance, replaceability and design for failure of containers. ReplicaSet is defined with a selector which allows to specify which are the Pods to be controlled, a number of replicas which indicates the number of Pods that should be maintained and a Pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet ensures that the given number of Pod replicas run at any time for the referenced Pod or group of Pods. ReplicaSet is a fundamental component in K8s platform, however it is rarely used directly. Instead, it is managed by a higher-lever component named Deployment, which allows to provide declarative updates for Pods and ReplicaSets: i.e. a desired state is specified in the Deployment definition, and the K8s Controller changes the actual state making it converge to the desired one. ReplicaSet is the component which allows to adopt a sort of design for failure, which means develop an application in such a way to take into account that the Pods it runs in can fail or be rescheduled at any time and, taking this into account, achieve more flexibility. Besides deployment-related aspects, Service is the component which offers the necessary functionalities to expose the application executed in a set of running Pods. This abstraction decouples the exposition of a service to external world from the actual execution of it. Hence, Pods that actually run the application can be destroyed, replaced or change their address during the service lifecycle. Service component offers the possibility to expose a stable access point to the actual service, independently from Pods-related aspects like rescheduling. ConfigMap and Secret are two useful components that are used for management of configuration files injection inside Pods. These can be passed via environment variables to the Pod or mounted as files. They have exactly the same function, the only difference is that Secrets are encoded in base64 and, starting from several versions, encrypted, so they are more suitable to use with sensitive information such as passwords or keys.

3.4 Virtual Machines vs Lightweight virtualization

After VMs and containers along with related platforms have been introduced, it is due to make a comparison, considering several points of view, to understand which is the most suitable solution in terms of vNSFs. When comparing these two types of solution, and in particular if the focus is

on VNFs, one of the most important aspects to take into consideration is the performance. The first aspect to investigate when talking about performance is the ease of deployment. Containers offer a much more rapid deployment time with respect to VMs. This is due to the fact that a container does not need its own virtual hardware to run and it shares the OS kernel with the host OS. This, combined to the fact that a container is usually much less heavy, in terms of occupied space, than a VM, since it is usually constructed in such a way it has only what it needs to run inside it, implies that the deployment of a container is much more cheaper in terms of CPU cycles and memory. On the other hand, when deploying a VM, all virtual hardware-related aspects, as explained in the previous section, have to be taken into consideration, introducing a non-negligible overhead in the deployment time.

Secondly, performance means also how much time does it need to complete an operation, once a container or VM is deployed. Also for this aspect, containers appear to be better than VMs. Banally, the motivation is that containers are not affected by the overhead introduced by the hypervisor at each privileged instruction execution. Instead, containers share the same kernel of the host machine, so when a privileged instruction is executed, the overhead introduced in terms of time is not much more significant than a traditional context switch. In terms of space, containers show a small memory footprint since the host OS instance is shared across all containers running on the same machine, whilst VMs show a much more heavy footprint since they require a standalone OS to be installed on them.

To provide an idea, comparing the performance offered by Docker containers and VMs, hosted on the same physical server, in the deployment of a simple web application composed by a Joomla [13] php front end server and a postgreSQL [14] database server as back end server, it comes out that Docker is able to reach approximately 5 times better performance than VM. In fact, comparing the number of requests that the servers were able to process in 10 minutes, using a test scenario in which the number of concurrent requests are increased linearly, starting from 1 and reaching about 80 concurrent requests in less then 9 minutes, it can be shown that containers are able to handle more than 5 times the number of requests processed by the VM. In addition, it can be shown that the VM takes also more time to process a single request. In the same test scenario, it can be demonstrated that after about 400 seconds, the VM responds to requests with a time about 5 times higher than containers [28].

In the context of vNSFs, performance aspects result to be crucial, due to the intrinsic vital importance of these type of functions. As an example, virtualized firewalls must guarantee performance also in presence of really varying workload. If the virtualization introduces too much overhead in such a way to make the user experience worse, obviously virtualization becomes no more convenient. In a context in which the workload can vary even significantly, containers turn out to be a more suitable choice for vNSFs, in particular for their ease of deployment and the performances that are succeeded to reach when scaling operations are needed, suddenly to a workload significant change. Moreover, it is due to mention that there are security functions that do not need much to use kernel data structures or features. As an example, an automated e-mail analyzer which is responsible to inspect message contents and classify them as spam or not, does not need to interact with the OS at all. For this reason, for a matter of memory usage, containers turn out ot be a better choice, since they do not need to occupy memory resources for an OS that is not even used.

Another important aspect when considering vNSFs is indeed the intrinsic security of VMs and containers themselves. If containers appear to be a much more better choice for performance-related aspects, they are cause for concern when considering security. This is again related to the fact that containers share the same host OS. This intrinsic aspect of containers is the cause of a reduction in isolation strength. As isolation reduces, isolated components are more interconnected and dependent on other components from the security point of view, and the security is bound to decrease exponentially. Since containers hosted on the same machine share the OS and kernel, for a potential attacker it could be easier to access all containers. For example, an attacker who

[13] https://www.joomla.org/ [Accessed: May 20, 2023]

[14] https://www.postgresql.org/ [Accessed: May 20, 2023]

gained a root access on a Linux machine can potentially control all the containers running on that machine. This would not be possible if VMs, each one necessarily with its own OS, were used. Platforms like Docker introduce improvements from this point of view, isolating many aspects of the underlying host from an application running in a container. Still, the separation introduced is not as strong as if VMs were used.

Performance and security are the two most significant aspects to take into consideration when evaluating virtualization solutions for vNSFs. However, other boundary aspects can be evaluated, as maintainability. For maintainability-related aspects, containers turn out to be a more convenient choice, due to the modularity introduced by microservice-oriented development. To maintain a software product usually means installing updates and security patches to correct vulnerabilities. In the context of a complex application, composed by several microservices, the update process can be related to the single microservice which needs it and, consequently, just a subset of the containers will be interested in it and the other can continue running, without interrupting the overall application functioning. On the other hand , when using VMs, this fine-grained modularity is lost and each update operation needs the whole application to be stopped and restarted, causing a service interruption. This can be particularly important when considering vNSFs. Considering the sample e-mail analyzer vNSF introduced above, each sub-component can be maintained independently and operations such as updates can be performed only on the interested sub-component without stopping the whole vNSF.

Finally, other aspects as portability must be considered. In some cases, applications still might be built to run on a specific operating system, for a matter of dependencies or similar aspects. Different operating system is not possible in containerized technology, whilst it is one of the strength points of using VMs.

Lastly, there is not an absolute better choice between containers and VMs when dealing with vNSFs. The former shows proven better performance and add less overhead, whilst the latter guarantees a better level of security since it provides stronger isolation. Moreover, it is due to cite that, while performance can be measured adopting precise metrics, like memory occupation or CPU cycles, the security cannot be evaluated in such an exact way. In addition, in the recent years, the world of research is deepening container security, presenting always more effective techniques to overcome the potential weaknesses introduced by the lower level of isolation. In conclusion, because of the better performance and resource utilization that they can guarantee and the continuous steps forward in managing their security, containers turn out to be an interesting alternative to VMs, and maybe they will be the way to go for virtualization of network security functions. In spite of everything, VMs remain a perfectly legitimate choice, despite the increased overhead that goes into performance, especially because of the isolation degree they can provide due to the presence of virtual hardware layer.

Chapter 4

Machine Learning

In this chapter machine learning will be presented. First, Section 4.1 provides a rapid introduction about what machine-learning is and how it generally works. Secondly, Section 4.3 will explain classification models, presenting general classification-related aspects and mechanisms. Then, Section 4.3.1 will present confusion matrix and metrics for classification models evaluation: methods useful for the evaluation of a model performances. Subsequently, two specific classification methods, particularly relevant in the context of this thesis work, that can be adopted for the construction of a model that is able to distinguish and divide data into several classes will be presented in Section 4.3.2. In particular, these methods are decision tree and random forest. Finally, Subsection 4.2 contains a brief explanation of overfitting and underfitting machine learning typical phenomenons.

4.1 Overview

One of the computer science fields that has accelerated strongly over the last few years is indeed machine learning: a branch of Artificial Intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving performances. AI is defined as "the science and engineering of making intelligent machines, especially intelligent computer programs"[1]. It can be related to the task of using machines to understand the human intelligence, but AI actually goes beyond this. Moreover, mistakenly, sometimes it is thought that the goal of AI is to create something that simulates the human intelligence, but this is not always nor usually true. On one side, It is true that humans can learn something about how to make machines solve problems by observing their own methods, with the aim of automating certain tasks. On the other side, from a certain point of view, most AI work is actually to study problems that the world presents, not only relying on human-like intelligence processes, but also finding solution adopting methods that are not usually observed in human beings, or that require much more computing power than a human brain can achieve.

Machine learning instead, as aforementioned, is a subfield of the AI, defined as the capability of a machine to imitate intelligent human behavior, learning new things from observation of past data. Despite its widespread popularity is relatively recent, machine learning started appearing right after the middle of the last century. Back in 1959, Arthur Lee Samuel, recognized with the Computer Pioneer Award by the IEEE Computer Society in 1987, coined the term "machine learning" [29]. The first application of machine learning was to create a computer program able to play autonomously a checkers game. The fact of choosing a game as a scenario was not random, in fact, it was thought that a game provided a convenient vehicle for such a study, since, with respect to real life problems, many complications or details are abstracted away. Still, the game contains the basics of an intellectual activity in which the learning process plays a major role.

[1] http://jmc.stanford.edu/articles/whatisai/whatisai.pdf [Accessed: Jun 03, 2023]

The idea was to program computers to learn from experience, instead of detailing methods for problem solving in minute and exact details for each single particular situation.

Machine learning turns out to be a perfectly suitable tool to face this type of problems, in which the whole solutions space cannot be explored because of obvious insurmountable operational limits. By a way of example, to play a checkers game or to recognize if a certain image of a face corresponds to a given person or not, which is one of the most popular modern machine learning-based applications, are problems which cannot be solved exploring the entire space of solutions. To face this type of challenges, some form of intelligence, which is not intrinsic in computers, is required. In fact, if on one hand computers are extremely fast in performing calculations like arithmetical operations, on the other hand they completely lack the ability of making decisions.

In traditional programming, the computer executes code, written by a human being and mostly based on combined if-then structures, so that when certain conditions are met the computer performs predefined actions. Machine learning follows a completely different approach, in which an automated process enables the computer to solve problems taking decisions based on past data observation, mostly of the times completely without human intervention. This automated process is called machine learning algorithm.

First, to make a model learn to solve problems and take certain decisions, a so called training phase is needed. During this stage, the algorithm is fed with past data it is able to observe and make calculation on, with the goal of identify patterns and consequently "learn". In this context, two macro-categories of machine learning types of algorithms can be distinguished: supervised and unsupervised learning.

Supervised learning algorithms and models learn to make predictions based on the observation of past labeled data [30]. A labeled training set (this is the name of the data used for the training process) means that each sample is composed by the input data and the associated output. A supervised learning model observes this data, makes calculation on it and finally make deductions based on what it saw, i.e. it learns to guess the label to be associated with data it has never seen before.

As a "toy example", consider an image recognition system which aims to recognize if an image contains a cat or not. To train such a system, the model must be fed with examples of images, each one with an attached label which classify them as "yes" (image contains a cat) or "no" (image does not contain a cat). The aforementioned example is actually just one of the two typical types of tasks of supervised learning, which is named classification, and the output consists in a label to assign to the input data. This kind of task consists in assigning the input an output label chosen from a predefined finite number of options. In addition, using supervised learning is also possible to perform another task, named regression. This second type of task consists in the prediction of some quantities, such as the number of sold items for a given category in a given period of time. Therefore, the output of this task is a continuous number within a given range.

On the other hand, unsupervised learning models and algorithms follow a completely different approach and have a totally diverse purpose [31]. These models are used to discover "hidden" relationships and patterns in data. In this case, such data are not labeled, which means the desired output is unknown. Therefore, the model has to extrapolate inferences from data without any training or observation of the "right answers", but understanding some kind of context and circumstances.

By way of example, one of the most common tasks of unsupervised learning consists in grouping similar data together. It has to be noticed that the definition of "similarity" between data in this context is not specified at all and the model itself has to extrapolate and guess parameters to compare data and understand if two sample are similar or not. This kind of activity is named clustering and finds application in many fields, such as e-commerce customer segmentation. As an example, getting behavioral data of customers as input, an unsupervised machine learning model may be able to identify groups of customers with similar behavior, providing aid to marketers.

In conclusion, machine learning, of whatever type it is, finds application in many fields, even very heterogeneous each other. The idea of enriching computers with a form of intelligence, started almost a century ago, is becoming really widespread in the recent years and it is one the new frontiers of automation.

4.2 Underfitting & overfitting

As briefly introduced in Section 4.1, supervised machine learning algorithms and models learn to make predictions based on the observation of past labeled data. In this context, an obvious deduction could be that the performance of the model strongly depends on the size of the dataset. In fact, the more correct examples are available to feed the algorithm with, the more it will understand and learn particulars to make better-quality predictions.

Actually, while this is true, since the size of the training set is crucial, on the other hand there are more aspects to take into account, related not only to the quantity, but to the quality of data used for the training. In fact, using poor-quality data when training machine learning models could introduce problems, which, in particular, consist in phenomenons that stake the name of are named underfitting and overfitting [32].

Underfitting[2] is said to occur when a machine learning model does not perform well enough, and fails to understand the relationships that bind a given input to a given class of output, generating many wrong predictions on both the training set and unseen data. Essentially, underfitting occurs when a model is too simple. When a model is underfitted (i.e. affected by underfitting), it simply is not able to recognize patterns in the data, which results in errors during the training phase and poor performance of the trained model.

Two possible causes that can lead to underfitting may be using a too small volume of data for training the model or using an insufficient amount of features to make the model understand the differences between data of different classes and consequently learn to distinguish. Therefore, possible corrections for the underfitting problem could be using a bigger quantity of data for the training phase, or the use of more sophisticated data with the presence of more features. Underfitted models are usually easy to identify, since it normally shows low values for the metrics, clearly indicating that the model is under-performing.

The opposite situation with respect to underfitting[3], is represented by overfitting. Overfitting is manifested when the model has been trained too much or when it contains too much complexity, which leads to bad performance when dealing with unseen data. This problem occurs when a machine learning model is "trained too well" on a specific dataset, resulting in the memorization of also noise and peculiarities of the training data.

In this way, model performs really bad on unseen data, which obviously do not show the same noise or peculiarities, since it results to be unable to generalize. The ability of a model to generalize is crucial and is what really distinguish machine learning from traditional programming. If a model cannot generalize well to unseen data, it cannot be relied upon for real problems.

Identifying overfitting can be more difficult than underfitting, because it is likely that the overfitted model performs really well on the test set extrapolated from the dataset and, unlike underfitting, metrics values are good, or even excellent, for the overfitted model. Moreover, when training machine learning models, it is more common to run into overfitting problems than to encounter underfitting.

Often, identifying overfitted models may not be an easy task, but in general it is possible to detect this problem testing the trained model, and calculating metrics values, on both the training and the test set. If the model performs much better with the training set than it does with the test set, then is likely that the model is overfitted. By way of example, consider the accuracy metric. If the accuracy calculated on the test set is really low, then the model is obviously underfitted. On the other hand, if the accuracy calculated on the test set is good overall, but significantly lower than the one calculated on the training set, then there is probably overfitting. The goal is actually to make the model be into the "good-fit" zone, where the accuracy of the model is good overall and shows not too different value for training and test set.

[2] https://www.ibm.com/topics/underfitting [Accessed: May 24, 2023]

[3] https://www.ibm.com/topics/overfitting [Accessed: May 24, 2023]

4.3 Classification models

As aforementioned in Section 4.1, classification is one of the typical and most important task when dealing with machine learning. Optical characters or speech recognition, spam filtering or the detection of a specific subject in an image are all examples of features that nowadays is usual to find in devices and software products. By way of example, modern e-mail clients that are able to classify a given suspect message as spam and raise an alert to the user, or mobile devices, such as smartphones, that are able to provide a translation on the fly of a text framed by their camera, can be mentioned. At the basis of the aforementioned functionalities and many other ones offered by modern software and devices, there is the capacity to solve a classification problem.

A classification problem consists in classifying a given input object in one of N possible classes, basing the decision on some kind of metric that represents the similarity of the input object properties with those of each class. In this context, a class is defined as a group or collection of similar object and the properties of an object are named features, while the concept of similarity between objects is defined by means of algorithmic distance functions and it is inversely proportional to it: the further apart two items are algorithmically, the less similar they are.

As a toy example, consider the problem of understanding if a given input fruit is an orange or not. In this case, the possible classes would be "orange" and "not orange", while the features could be the color, the diameter, the shape or any other characteristic that makes an orange differ from other kind of fruits. The similarity of a given input fruit is quantified by calculating the distance of each one of its features values with the orange's typical ones. If the distance is under a given threshold, than the input fruit is classified (labeled) as an orange, otherwise it is not.

In many cases, when dealing with problems that are much more complex than the presented toy example, understanding which are the most suitable features to correctly and fully represent the properties and characteristics of the objects belonging to a given class, is not a trivial task. Moreover, additional problems derive from the the usage of high-dimensional data, which means data represented by many features, that can adversely affect the performance of the model leading to overfitting problems, which will be presented in Section 4.2, or introducing immoderate storage requirements [33]. For these reasons, many times, a data preprocessing strategy, known as feature selection or feature extraction, is needed. Feature selection plays a crucial role in optimizing the performance of classification models and it involves transforming raw data into a more compact, informative and understandable representation, enabling the models to effectively capture relevant patterns and make accurate predictions [34].

Actually, the above presented fruit classifier toy example is just one type of classification problems, named binary classification. A binary classification problem objective is to classify data into two classes, like "orange"/"not orange", "yes"/"no" or "good"/"bad". Actually, one more macro-category of classification problems exist and is referred as multi-class classification. The core logic is the same but, recalling the previous toy example, here a possible problem could be to recognize an input fruit among 10 possible different kind of fruits. From the used features point of view, there would be no changes with respect to the previously presented problem, but here the distance of the features values of the input fruit would be calculated for each one of the possible classes. At the end, the smallest distance gives the class of the input element. Solving a multi-class classification problems means, in point of fact, to classify data into three or more classes.

Machine learning objective is, in general, to make machines emulate the way the human beings think and learn, as briefly discussed in Section 4.1. Hence, to fully understand how a machine learning model works, it is first necessary to figure out which is the process of learning in a human being. Especially during the childhood, when the human brain assimilates a huge quantity of information, the infant starts a learning process based on the observation of what he is able to see and, in case of human beings, also smell, ear, taste and feel. Placing the focus especially on the sight, it is possible to affirm that machine learning models follow a really similar process of learning, based on the observation of data. Recalling the usual example, the fruit classifier understands how to properly classify fruits by observing a set correct examples. This process is called training and it represents the core activity of machine learning. Actually, the presented process is referred as supervised learning, since the training process is based on the

observation of correct example. Classification is a supervised learning task and follows this mode, while other tasks follow a different learning process based on the observation of unlabeled data, namely referred as unsupervised learning. These tasks consists more in the discovery of hidden correlations between data.

As aforementioned, the prediction made by the model is based on algorithmic distance measurements to understand which class the input object is most similar to. This means that, considering classifiers, during the training phase, the machine learning model goal is to set a threshold which well separates objects belonging to different classes, so that, when an unseen input arrives, the algorithm can classify it based on its features values and considering if the input is beyond or on this side of the threshold. Practically, this threshold is represented by a function of $N-1$ variables, with N being the number of features considered, and it is calculated and set optimizing this function. This threshold takes the name of decision boundary. To better understand this process, consider the example shown in Figure 4.1. The example illustrates the features plot for a two-featured (x_1 and x_2) binary classification problem, with "Cross" and "Circle" being the possible classes.

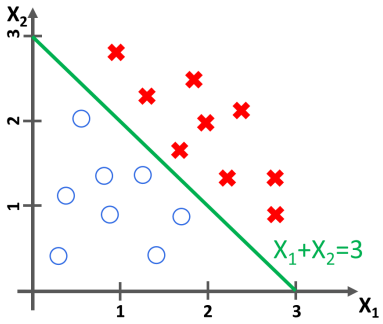

Figure 4.1. Example of decision boundary for a two-featured binary classification

In this trivial case, the classification algorithm objective consists in finding the threshold which better separates the items belonging to class "Cross" and "Circle". In particular, this threshold is represented by the line $x_1 + x_2 = 3$. Setting this threshold, the classification algorithm will predict "Cross" when $x_1 + x_2 >= 3$ and "Cirle" when $x_1 + x_2 < 3$. It is due to cite that the decision boundary does not necessarily need to be linear but can have much more complex shapes that are also difficult to represent graphically.

The data observed by the model during the learning process take the name of training set. Hence, in the context of classification and supervised learning in general, a dataset is defined a set or collection of correct examples used to teach the machine learning algorithm how to make predictions, through the observation of it. To take full advantage of machine learning it is crucial to feed the model with an appropriate dataset, constituted in such a way as to optimize the learning process. There are no absolute rules to follow when choosing or building a dataset to ensure it is suitable, but some important empirical properties can help:

- high accuracy: refers to the correctness level of data. More specifically, it refers to how well the available data represent the situation. For example, recalling the usual toy example, if some of the fruit diameter is given in kilometers or is negative, the information is not accurate;
- reliability: is an important qualitative attribute which ensures the absence of contradictions and the coherence of the information. A model trained on reliable data is more likely to make correct predictions than one trained on unreliable data. A reliable training set should not include duplicates or records with missed or omitted values;
- consistency in feature representation: is an important characteristic which ensures data compatibility. For example, some machine learning models require transformation of non-numeric features to numeric, or converting the actual range of values to a standard range.

- right size: is an extremely important characteristic that influences and even be a bottleneck for the overall quality of the predictions. There is not an absolute rule to correctly identify the dataset size, and it strongly depends on the problem addressed.
- diversity: is a crucial factor when considering the quality of a dataset. Data must be "representative" enough to represent all the facets of the problem that the model will have to solve. Consider the usual toy example as if the classifier worked with images instead of raw data. The training data for the model should be images of different fruit captured, for example, from different angles, under different lighting conditions and from varying distance. In this way it is more likely that the model can classify fruit more accurately as compared to if it is trained on a model which has all similar images.
- completeness: refers to the comprehensiveness of the data. This ensures that each important and relevant facets of information is used for the model training.
- relevance: refers to how significant the available data is to solve the problem. Even if it could seem trivial, if the dataset contains information that are unrelated or irrelevant to the problem addressed, the quality of the predictions by the model will not be as expected.

Actually, when training a machine learning model, the whole dataset is split in two parts, namely training and test set. The training set is the subset of correct examples which is observed by the model and is used to train it. On the other hand, the test set is the subset of data, not used during the training phase, which is used to evaluate the goodness of the training. In this context, it is important that the training and test set are disjointed, that is, they have no elements in common. This is because including elements belonging also to the training set in the test set would likely lead to "optimistic" results in the evaluation of the model, since the test set would contain elements already known to it.

In this way, a small portion of the dataset is used as a collection of correct examples for the purpose of assessing the model's ability to make correct predictions. During the test phase, the model is fed with the test set records, without providing the correct class associated to these records. Subsequently, the prediction made by the model for each input record is compared to the correct class associated to it. In this way it is possible to evaluate the quality of the predictions made by the model, after the training phase.

Finally, it is appropriate to briefly discuss how the output is represented. For both the binary and multi-class classification problems, the solution can be represented in two different ways, namely hard assignment and soft assignment. When following the hard assignment approach, the input data is classified in just one of the possible classes. On the other hand, when using soft assignment, the output is given by the probability with which each possible class matched the data. Therefore, recalling the usual toy example, if the problem is to categorize one fruit among 10 possible classes, using the hard assignment approach, the output representing the solution of the problem will consist of the fruit category that more matches the input fruit characteristics. On the opposite, following the soft assignment approach, the solution of the problem will be represented by a vector which contains, in each position, the probability that the class in that position is the right one for the specific input fruit.

4.3.1 Classification metrics

The evaluation of the training phase goodness, made through the observation of the predictions carried out by the model on the test set data, is actually done calculating several quantitative metrics representative of the correctness of these predictions. One of the most important techniques of evaluation is given by the confusion matrix [35]. The confusion matrix is a table which represents the number of correct and incorrect prediction for each possible output class. The confusion matrix shows a row and a column for each class. In particular, columns represent actual values, while rows represent values predicted by the model during the test phase.

The sum of the values along the main diagonal indicates the number of good predictions, while the values out of the main diagonal represent the number of wrong predictions made by

the model. Generally, considering N classes, the number of good predictions is given by the sum of True Positive (TP) and True Negative (TN) predictions, respectively, the number of items of class "i" correctly classified as "i" and the number of "not i" items, correctly classified as "not i". Instead, the number of wrong predictions is given by the sum of False Positive (FP) and False Negative (FN) predictions, respectively, the number of "not i" items classified as "i" and the number of class "i" items classified as "not i".

In case of a multi-class classifier, TP, TN, FP, FN must be calculated for each single class. Recalling the usual toy example, assume that the confusion matrix is intended to evaluate a classifier which must recognize between three different classes of fruits, namely "Orange", "Apple" and "Pear", as shown in Figure 4.2. Taking as reference the class "Orange", in this case, the sum of values of a column "Orange" represent the total items truly associated to the "Orange" class, while the sum of values of a given "Orange" row represent the total number of items labeled as "Orange" by the model. Also here, the sum of the values along the main diagonal indicates the number of good predictions, while number of wrong predictions made by the model is given by the sum of values out of the main diagonal. Also here, the number of good predictions is given by the sum of True Positive (TP) and True Negative (TN) predictions, which is, respectively, the number of items of class "Orange" correctly classified as "Orange" and the number of not "Orange" (i.e. "Apple" or "Pear") items, correctly not classified as "Orange". The number of wrong predictions is given by the sum of False Positive (FP) and False Negative (FN) predictions, which is, respectively, the number of not "Orange" items classified as "Orange" and the number of class "Orange" items not classified as "Orange".

		ACTUAL VALUES		
		Orange	Apple	Pear
PREDICTED VALUES	Orange	TP	FP	FP
	Apple	FN	TN	TN
	Pear	FN	TN	TN

Figure 4.2. Confusion matrix for a multi-class classifier

From the confusion matrix and concepts of TP, TN, FP and FN it is possible to extrapolate some interesting metrics, which are useful to understand and evaluate how the model performs. Missing evaluation of the machine learning model via different metrics can lead to unexpected problems in the performance, intended as correctness in the predictions, when the model faces unseen data. Some of the most important metrics can be extracted observing the confusion matrix.

In general, when considering multi-class classifier, these metrics derive from the correspondent ones for binary classification problems [36]. Since these metrics give indications referred to a single class with respect to the others, in the multi-class case, an average of the metrics values obtained for each individual class is considered, applying the concept of micro-averaging, and macro-averaging. Micro-averaging favors classes represented by more items, while macro-averaging treats all classes equally. Useful metrics for evaluation of a multi-class classifier are described in Table 4.3.1, where N is intended to be the number of classes and M refers to macro-averaging.

It is due to mention that, as there is yet no well-developed multi-class ROC[4] analysis, AUC[5], which is one of the most effective metrics for binary classification, is not included in the list of multi-classification metrics. However, it is due to cite that, while accuracy, precision, recall,

[4]Reception Operating Characteristic curve, created by plotting the value of sensitivity versus specificity.

[5]Area Under Curve, is defined as the area which is enclosed under the ROC curve.

Table 4.1. Metrics for multi-class classification model evaluation

Metric	Formula	Description
balanced accuracy	$\frac{\sum_{i=1}^{N} \frac{TP_i+TN_i}{TP_i+TN_i+FP_i+FN_i}}{N}$	is calculated as the summation of sum of two correct predictions divided by the total number of dataset entries, all divided by the number of classes. Also known as average accuracy. The best reachable accuracy value is 1.0, and the worst is 0.00. It provides the average per-class effectiveness of a classifier
precision$_M$	$\frac{\sum_{i=1}^{N} \frac{TP_i}{TP_i+FP_i}}{N}$	is calculated as the summation of correct positive predictions (TP) divided by the number of positive predictions (TP + FP), all divided by the number of classes. The best reachable precision value is 1.00 and the worst is 0.00. It indicates the agreement of the data class labels with those of a classifiers, treating all classes equally
recall$_M$	$\frac{\sum_{i=1}^{N} \frac{TP_i}{TP_i+FN_i}}{N}$	is calculated as the summation of accurate positive predictions (TP) divided by the total number of positive (P), all divided by the number of classes. Also called sensitivity or TP rate. The best reachable recall value is 1.0 and the worst 0.0. It provides an indication on missed positive predictions, treating all classes equally
F score$_M$	$\frac{(\beta^2+1)precision_M recall_M}{\beta^2 precision_M + recall_M}$	is a measure of the accuracy of the test. It is calculated by the formula: $\frac{2*precision*recall}{precision+recall}$. It is an aggregate metrics which symmetrically provides indications on precision and recall in a single metric, treating all classes equally

and F-score are influenced by the unbalance of the data set and can be misleading (especially if there is a class with lots of samples with respect to the others), AUC and average accuracy are independent from the unbalance, which makes them more truthful metrics in this type of scenario.

4.3.2 Decision tree and Random forest

After having provided a general explanation on how a classification machine learning model works, it is due to present two specific classification algorithms, which have been relevant in the context of this work. First, it is due to mention that there is not an absolutely better algorithm with respect to the others, but the performance, here intended as both prediction correctness and optimization of resource usage, strongly depends on the specific dataset characteristics and the specific problem treated.

Moreover, each classification, and in general machine learning, algorithms performance can be really influenced by its hyperparameters values. Each machine learning model is defined by a set of model parameters, which are tuned during the training phase. Besides model parameters, each algorithm offers also a set of hyperparameters, which are defined as "top-level" parameters whose values control the learning process and influence the values of the model parameters, and can be defined by the practitioner when configuring the model. For the example of multi-class classifier, one of the provided hyperparameters is usually the "class weights", which could be useful to instruct the model about the classes unbalance, which means that some classes have much more objects than others.

Decision tree

One important classification method, which finds its computational origins in models of biological and cognitive processes, is the decision tree [37]. Decision trees, sometimes referred to also as classification or regression trees, are general purpose prediction and classification methods, among the first to be electronically implemented in digital circuits in the later decades of the last century. A decision tree uses a hierarchical tree structure, organized through a root node, branches, internal nodes and leaf nodes.

Edges arising from nodes labeled with a feature are categorized with possible subset of values of the feature itself. Each leaf in the tree is labeled with a class or probability distribution over the classes. In essence, a decision tree classifies items by posing a series of questions about the features associated with the items. Each question is contained in a node, and every internal node points to one child node for each possible answer to its question [38].

Recalling the usual toy example used in this Chapter, an hypothetical decision tree which aims to distinguish oranges among a collection of input fruits, could pose questions related, for example, to the shape, the color and the weight of the input fruit, as represented in Figure 4.3. In this case, leaves of the tree are labeled with a class and not with probability. Generalizing, classes may be represented by any type of measures, e.g. nominal, ordinal, or interval. When nominal targets are used, as in the case shown in Figure 4.3, the tree is referred to as a "classification tree".

The branch partitions are based on a selection that is taken from a search through the available data, to discover features that can be used as partitioning criteria to best describe the variability among the possible classes. Trivially, It is normal to select the feature that produces the most significant separation in the variability among the descendant nodes.

In Figure 4.3, the first level of the decision tree is produced by selecting the "Color" feature, so it has to be assumed that the algorithm, observing the available data, discovered that, primarily, the "Color" feature is the best to distinguish "Orange" and "Not Orange" objects. Obviously, the selection of the best feature is not so trivial when dealing with real-life scenarios, and it is an open subject for the research world, since decision trees allow for a variety of computational approaches to feature selection and the choice of the first feature strongly influences the decision tree performance.

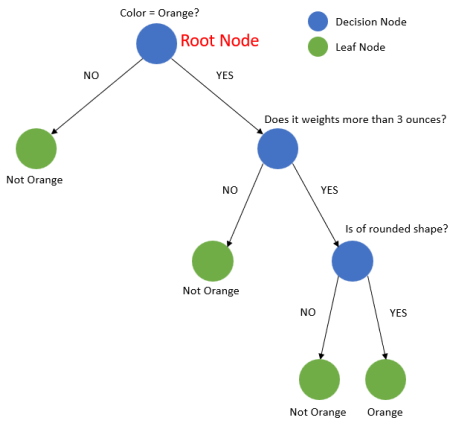

Figure 4.3. Decision tree example

Actually, the example case depicted in Figure 4.3 represents one of the most optimistic situations in the construction of a decision tree. In fact, assuming to have just two classes, the ideal binary test is one for which an answer of "yes" means the example is always in one class and an answer of "no" means the example is always in the other class [39]. In a case like this, using these ideal binary tests at each node, the model will get perfect classification accuracy. In essence,

recalling the usual trivial example, it is impossible that the model will misclassify as "orange" a fruit which is not orange in color, weights less than 3 ounces or is not rounded shape.

In this context, the choice of the best test to use at a node represents the first and perhaps most interesting problem in designing a decision tree algorithm. However, such tests are hard to find, and sometimes they may not exist at all. Thus, the problem at each node of the decision tree consists in finding the test which better "separates" the items and to come as close as possible to the ideal case.

Therefore, the decision of the test to be performed in a given internal node is not trivial, and various algorithmic solutions have been proposed. In general, the construction of a decision tree follows a "divide and conquer" approach [39]. Given a set T of training samples, the basic idea is to refine T at each step into subsets that are, or are close to be, single-class collections of samples. At each node, a given test based on a single feature is selected. This test will present a set of N mutually exclusive outcomes $\{O_1, ..., O_i, ..., O_N\}$. Then, T is partitioned into subsets $\{T_1, ..., , T_i, ..., T_N\}$, where each subset T_i contains all the cases in T which has outcome O_i for the selected test.

After the first separation criteria is defined, the same tree-building logic is repeated recursively for every descendant node, which means, for each new subset of training samples. Any test divides the subset of training samples T in a non-trivial way, which means each one of the resulting T_i is not empty, and will eventually result in a division into single-class subsets, even if many or all of them contain a single training sample.

However, a significant number of samples in each leaf node is required to make the tree acquire predictive power. The ideal case consists in the choice of a test at each internal node so that the resulting tree is small and, likely, the resulting partitions in the leaf node have good representation in terms of training samples. Unfortunately, the problem of finding the most optimized decision tree for a given training set is NP-complete[6]. Therefore, most decision tree building methods consist in greedy algorithms[7].

Consequently, a criterion is needed to select the best test at each internal node, which, for most of the decision tree algorithms, is based on the concept of information gain. The information theory at the basis of this criterion can be summarized as follows: "the information conveyed by a message depends on it probability and can be measured in bits as minus the logarithm to base 2 of that probability" [39]. Adopting the notation $info(T)$ for the information conveyed by a given subset of samples, and assuming that T has been partitioned in accordance to the n results of a test X, the expected information can be calculated as

$$info_X(T) = \sum_{i=1}^{n} \frac{|T_i|}{|T|} info(T_i)$$

where $|T|$ represents the cardinality of the set T.

Consequently, the quantity

$$gain(X) = info(T) - info(T_i)$$

indicates the information that is gained by partitioning the input subset T in accordance with the specific test X. Essentially, the information gain criterion consists in the selection of the test which maximizes the information gain quantity consequently to the split.

By default, the algorithm stops when all subsets contain samples belonging to a single class. However, to avoid overfitting, which has been presented in Section 4.2, it is possible to spot the split of the decision tree, and therefore end the algorithm, specifying a given stop condition [37]. Actually, the stop condition can be seen as the non-compliance with given split criterion on all nodes. A split criterion is a condition that has to be respected on a specific node to continue the branch growing, and therefore split that node. Examples of split criteria could be the minimum number of objects that an internal node must have or the minimum number of

[6]NP-complete problems are problems which cannot be solved by any polynomial time algorithm.

[7]A greedy algorithm is an algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.

objects that a leaf node must have, to proceed with the split. The mentioned, non-exhaustive, examples are hyperparameters of the decision tree and can be set by the practitioner to optimize the performance.

Random forest

Another extremely successful algorithm in this context is random forest. Random forest algorithm, proposed in 2001 by Leo Breiman, statistic professor at University of California, Berkeley, is a general-purpose classification and regression method [40]. Random forest consists in an ensemble of several decision trees and the prediction by the forest is made aggregating their results provided by the single trees via majority voting operations. The idea born from the necessity to have algorithms that scale with the quantity of information, to fully take advantage of the always growing dimensions of datasets.

In this context, random forest is one of the most successful methods available nowadays to manage large quantities of data. Overall, having already presented decision trees, the random forest logic can be described straightforwardly, following the simple but effective "divide et impera" principle: first, some subsets of data are built from the available dataset, then, a decision tree (in this context called estimator) is trained on each subset of data.

Recalling the decision tree construction process cited above, in case of random forests, each decision tree is constructed using a different subset of the training data, randomly selected with replacement, and a subset of the input features. It is worth remarking that, using bagging, the training data are not split into smaller chunks, training each tree on a different chunk. Rather, if the original dataset is composed by N samples, each tree is fed with a training set of size N, but instead of the original training data, a random sample of size N with replacement is used.

Recalling the usual toy example, if the training set consists in six fruits [fruit1, fruit2, fruit3, fruit4, fruit5, fruit6], then an example of training set for one of the trees could be provided by the randomly generated list [fruit1, fruit1, fruit3, fruit4, fruit4, fruit5]. In essence, the concept of replacement consists in the fact that the same training sample could be involved in the construction of more than one decision tree.

The randomness introduced reduces the correlation among the trees and promotes the overall diversity, which means, the construction of decision trees that are too similar is avoided. In such a way, the forest will be composed of decision trees which "protect each other" from errors, since, being constructed using completely different and randomly selected data, while some trees may be wrong, many other trees will be right, so, as a group, the trees are able to move in the correct direction, obviously until all trees does not error in the same "direction".

In addition, as aforementioned, each individual tree is assigned a subset of features. In a traditional standalone decision tree, when a node has to be split, every possible test based on a single feature is considered and the one that produces the most separation between the observations in child nodes is picked. On the other hand, each tree in a random forest can consider only on a subset of the available features to build the test. This introduces more variation among the trees and finally results in lower correlation across trees in the forest and more diversification, contributing to improve overall performance [41].

Considering the usual toy example, assume four features are used to characterize the fruits: "color", "weight", "width", "height". A standalone decision tree, when splitting a given node, will select the test based on the feature which maximizes the information gain: let's assume it is the feature "color". On the other hand, assuming a random forest composed by two decision trees $T1$ and $T2$, and assuming features "color" and "width" assigned to $T1$, and features "weight" and "height" assigned to $T2$, $T1$ will be able to select the global better test (based on "color" feature), while $T2$ will choose the local best test, based on "weight" or "height". This contributes to the construction of completely different and uncorrelated trees.

Therefore, the random forest is composed by trees that are not only trained on different sets of data (thanks to bagging) but also use different features to make decisions and this creates uncorrelated trees that buffer and protect each other from their errors.

Finally, when a prediction has to be performed, the resulting predictions from each estimator are aggregated together, using, in case of classification, the majority voting technique: the class predicted by the most single decision trees is the one provided in out by the random forest. A minimal graphic example of the process is represented in Figure 4.4. The whole process presented takes the name of bagging, and it represents the core logic of the random forest algorithm.

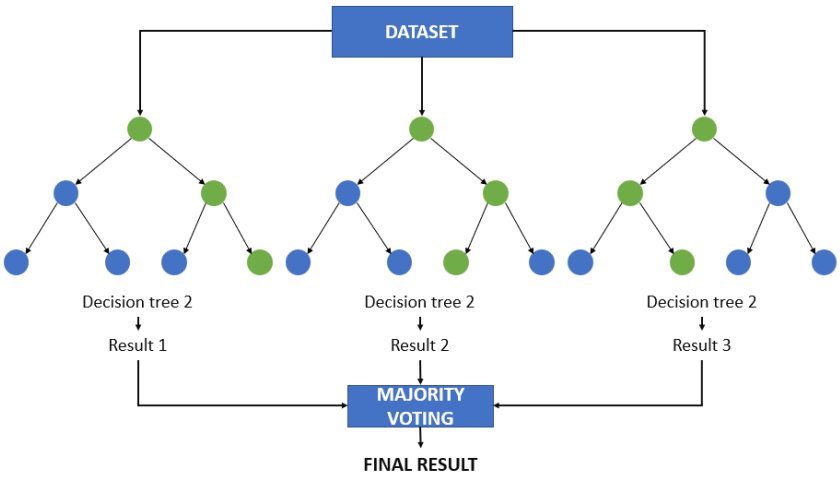

Figure 4.4. Random forest example

In conclusion, it is due to cite several advantages offered by random forests. One of the strong points of random forest is represented by the fact that they show good performances on a wide range of prediction problems, still having relatively a few parameters to tune. Moreover, additional strong points of this algorithm are the ability to measure feature importance, the intrinsic capability of dealing with multi-class classification and the ability to handle high-dimensional datasets, even in presence of missing data effectively. Moreover, due to the fact that the single decision trees are constructed individually, random forests result to be computationally efficient, since the whole process can be parallelized. Finally, it has to be mentioned that the choice of using random forest in this thesis work was largely motivated by the great performance obtained using random forest in solving really similar problems, such as the identification of the network tools which originate certain packets sniffed [42].

Chapter 5

Cybersecurity for IoT

In this chapter, cybersecurity-related topics, especially pertinent to the context of the present work, are going to being presented. First, Section 5.1 provides a brief introduction about the treated problem and sets the focus on the particular competence area: network security. Secondly, Section 5.2 explains the IDS/IPS technologies, important tools in the context of attacks detection and prevention, providing also a discussion about the usage of these tools in an IoT context, in Section 5.2.1. Moreover, a discussion about known IDS/IPS implementational solutions is provided in Section 5.2.2. Subsequently, another important technology in the context of network protection, namely the VPN, will be presented in Section 5.3, again providing a discussion about peculiarities of the usage of this technology in an IoT scenario, in Section 5.3.1, and presenting some widespread VPN terminator implementation, in Section 5.3.2. Finally, Section 5.4 provides a discussion about the attacks that should be mitigated by employing the previously presented solutions, in particular focusing on traffic eavesdropping, discussed in Section 5.4.1 and two additional types of attacks, namely "flooding" and "RTSP bruteforce", discussed in Section 5.4.3 and Section 5.4.4 respectively, and on another feature of the proposed solution, anomalous traffic detection, in Section 5.4.2.

5.1 Overview

Cybersecurity, a multifaceted realm of paramount importance, encompasses a wide range of disciplines aimed at shielding digital environments, but also processes and people, from threats and vulnerabilities. The global objective is not only the safeguard of sensitive data and critical systems and infrastructures, but also ensuring the continuity of operations, protecting against disruptive cyber-incidents.

When dealing with data and information, therefore referring to the more technical meaning of security, the objective is, in general, to guarantee the CIA (Confidentiality, Integrity, Availability) triplet of fundamental properties to this information. Respectively, these properties indicate the inaccessibility to information by unauthorized users, the capacity of detecting whether a given information has been modified while in transit from source to destination and the survivability of network services to authorized parties when needed.

Nowadays, IoT environments are becoming more popular. In the IoT, a thing can be anything on the planet: a person with a blood pressure monitor implant, a car endowed with sensors that alert the conductor when the tire pressure is low, a farm animal with a transponder, or any object that can be assigned an IP address and potentially transfers data over a network.

As introduced in Chapter 1, usually, the main cause of complexity when considering security for IoT devices networks is the limited amount of hardware resources mounted on these devices, for a matter of costs and sometimes device size, as well as the lack of standard implementations and of suitable security techniques. For all this series of reasons, while IoT systems are exposed, as well as traditional devices, to threats like network intrusion and/or malware, the main problem resides in the lack of standard solution in order to mitigate these threats.

Among all the cybersecurity fields, one of the most interesting, especially from the companies point of view, is indeed the network security, which represents also the main cybersecurity field treated in the context of the present work. The network security is one of the main concerns for companies which potentially also use IoT technologies, and its primary objective is dealing with the protection of the information, infrastructure and business continuity from the threats which make use of the network as attack vector. Adequately protecting the networks, trying to avoid threats coming from the external world, turns out to be essential for companies in order to preserve their information and business continuity.

The network protection is achieved thanks to the employing of a set of different network security tools, which encompasses, among the others, firewalls, IDS (Intrusion Detection System) and IPS (Intrusion Prevention System), VPN (Virtual Private Networks), access control and encryption mechanisms. In the context of the present work, two network protection technologies turn out to be particularly relevant: IPS and VPN.

The IPS, that will be deepened in Section 5.2, can be seen as an evolution of its predecessor, the IDS, and constitutes a proactive defense mechanism that aims to identify and tackle potential malicious activities. IPS solutions, through real-time analysis and automated response, contribute significantly to the overall network security posture, providing a layer of defense against emerging threats.

The VPN, that will be discussed in Section 5.3, is a cryptographic tool that creates secure and encrypted communication channels over public networks, ensuring the privacy and integrity of transmitted data. By establishing "virtual" connections between remote endpoints and using encryption protocols to encapsulate data in a secure tunnel, a VPN safeguards data from eavesdropping and unauthorized interception.

5.2 Intrusion Detection and Prevention Systems

The networks are incessantly facing diversified cyber-attacks types. Because of their particular nature, IoT devices are extremely prone to various threats and attacks, which makes them become very attractive targets for potential attackers. Addressing these challenges is essential for maintaining an indispensable level of security for IoT networks.

In this context, the ability to recognize an ongoing attack and potentially block it before the malicious activity can actually reach its end causing damage to the network or devices, becomes fundamental in order to protect confidentiality, integrity and availability of infrastructures and related data. In fact, besides the intrinsic weaknesses of IoT devices, introduced by the reasons presented in Section 5.1, one of the main problems of IoT devices networks is the lack of appropriate security measures from the network point of view [43].

In fact, given the nature of IoT devices, the protection of this type of devices at the network level is often overlooked, placing more attention on seemingly more critical systems and it is not uncommon to find, by a way of example, IoT systems that are connected to the network but without the necessary shielding, which could be provided by, for example, a firewall.

As it will be discussed in Section 5.4, one of the main concerns when considering IoT devices is the possibility to exploit the inherent weaknesses of these systems in order to take control of them, for example via malware infection, and use these devices within a botnet, in order to increase the firepower of a potential attacker. Consequently, it arises the necessity to establish appropriate network security measures with the objective to firstly detect and potentially prevent, avoid and block malicious activities.

Analyzing the solutions proposed in the literature for detecting and/or preventing attacks, it can be stated that the most common method of countering attacks coming from the network are firewalls. However, traditional firewalls, usually based on packet filtering therefore regulating packets flows inside a network based on source and destination addresses, ports or protocols, are usually not smart enough to detect and block some kinds of threats. In this context, two solutions, smarter than traditional firewalls based on packet filtering, can be identified, namely IDS (Intrusion Detection System) and IPS (Intrusion Prevention System). These two solutions,

which can be seen as evolving each other, aim to identify malicious activities through network traffic analysis in order to report anomalies, in the case of IDS, and additionally to try to mitigate the threat in the case of IPS.

IDS, in fact, is a precautionary measure where the system itself takes no action in case an intrusion/attack is detected, instead, this kind of tool is able to raise an alert when an anomaly is detected through the analysis of its monitoring scope, such as a network or a single system. In this respect, it is due to mention that several types of IDS are possible, based on where the IDS is installed, thus on the IDS monitoring scope, and on the specificity of the analysis it performs [44].

By way of example, ad IDS can be installed on a dedicated network device (physical or virtual) and be connected to the network. In such a way, the system is able to monitor and analyze all the traffic that passes through it. In this case, the IDS takes the name of NIDS (Network IDS). This solution guarantees scalability, since it is independent from the number of hosts connected to the network, while it lacks on specificity, since analyzing the whole network traffic in a generic way, it could happen that a threat was not detected. On the other hand, an IDS could also be installed directly on hosts connected to the network, putting in place an HIDS (Host IDS). An HIDS works by taking "snapshots" of the device assigned to it and, through the comparison of the most recent snapshot to past ones, the HIDS tires to identify the differences that could indicate a potential intrusion.

Differently from an IDS, as aforementioned, an IPS follows a more proactive approach and can be seen as an evolution of an IDS. In fact, in addition to detecting the threat by identifying malicious activity through network traffic analysis, an IPS is able to actively perform corrective actions, with the purpose to interrupt the ongoing malicious activity before it caused damages to the targets [45].

To provide a practical example, when considering DoS (Denial of Service) attacks, that will be discussed in Section 5.4, an IDS running on the network could be able to detect the ongoing attack through the analysis of the network traffic, noticing an abnormal amount of packets in transit, and then raise an alert that will trigger the human intervention to mitigate the threat. An IPS instead, additionally, in case it detects a similar situation, would be able to actively perform mitigative actions, such as blocking the source of the vast quantity of packets sent or reducing the bandwidth of the channel for traffic coming from that source, trying to avoid potential damages caused by such attack.

As it was mentioned for the IDS, also for the IPS there are different possible typologies. Similarly to IDS, based on where an IPS is placed and on the scope of network traffic it analyzes, it is possible to configure a NIPS (Network IPS) or an HIPS (Host IPS). Respectively, a NIPS is installed at the network perimeter and is in charge of the monitoring of all traffic that enters and exits the network, while an HIPS is installed on a single host inside a network and is responsible for the analysis of all the traffic that goes in and out of the related host.

Given their similar nature and similar purpose, with the only difference of active and proactive actions that an IPS is able to perform with respect to an IDS, it is possible to compare these two technologies, in order to have an overview of the advantages and disadvantages of each.

As it is demonstrated by Table 5.1, which contains a comparative analysis of IDS and IPS systems [43], both technologies show advantages and drawbacks. In particular, unsurprisingly, the differences come from the fact that while the IDS is a "passive" technology, which requires the human intervention for the mitigation of a detected threat, the IPS acts as an "active" component, which is able to directly perform automated actions, configured by a set of rules. Indeed, this possibility to take the respective countermeasures without human intervention is practically translated in a much lower response time to attacks and threats. However, on the other hand, this proactive behavior introduces criticalities when the reckon performed by the IPS is not accurate.

Another difference between these systems, not specified in Table 5.1, is their placement from

IDS	IPS
Detects threats and monitors the associated system	Monitors and protects the associated systems performing active mitigative actions
Only raises alerts/alarm, therefore human intervention is required for action	No human intervention is required since it takes action based on a pre-configured set of rules
False alarm rate does not directly impact performance	False alarm rate is high concern because of the proactive behavior
Legitimate traffic is not blocked	Legitimate traffic might be blocked due to wrong detection

Table 5.1. Comparison between IDS and IPS technologies

the network topology point of view, as explained by Cisco[1] and paloalto[2], two of the major network appliances vendors in the world. Considering a NIDS/NIPS, as it is illustrated in Figure 5.1, an IPS must be placed inline in a network topology, so that it is directly traversed by the network traffic flow between the source and destination and to have the possibility to take active actions. This requirement in network placement represents a further difference between IPS from its predecessor, the IDS. Conversely, an IDS is a passive system, so it can only scan and monitor network traffic and eventually reports back on threats. Being placed in parallel to the monitored systems and working on copies of the traffic, an IDS adds less overhead with respect to the IPS, since no additional hops to traverse are introduced to the traffic normal path.

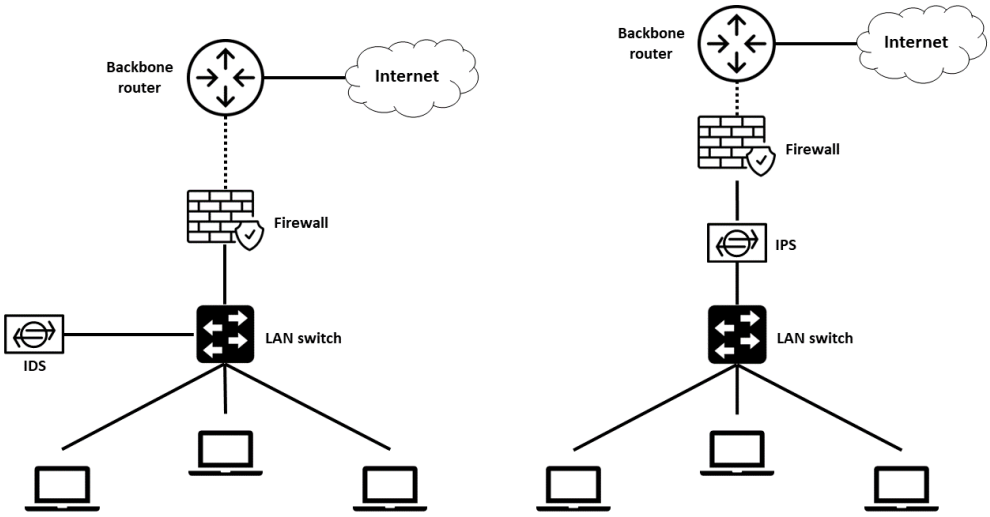

Figure 5.1. Example of network topology when deploying IDS and IPS

For the aforementioned reasons, an IPS must work efficiently to avoid decreasing network performance. Since an additional hop that must be traversed by network traffic is introduced when using an IPS, the performance aspect turns out to be crucial to not introduce discomfort in network communications, as usual trying to improve the overall security without affecting the functionality. Therefore, it is fundamental that an IPS works in an efficient way.

[1] https://study-ccna.com/firewalls-ids-ips-explanation-comparison/ [Accessed: Aug 25, 2023]

[2] https://www.paloaltonetworks.com/cyberpedia/what-is-an-intrusion-prevention-system-ips [Accessed: Aug 25, 2023]

Beyond the performance aspects connected with the temporal overhead added by an IPS, the accuracy performance of this system must also be considered. Recalling the Table 5.1 and the aforementioned example scenario in which an ongoing DoS attack is detected and assuming that the detection consists of a false positive prediction, in case of an IDS the only complication would be due to the presence of an unnecessary alert, which can also be totally ignored. On the other hand, when considering an IPS in the same scenario, the proactive mitigation performed could lead to disruption of operation, as legitimate traffic, maybe due to an application that requires to generate a big amount of network traffic, would be actively restricted automatically by the tool.

For this reason, it turns out to be essential that an IPS shows good performance in avoiding the detection of false positives. With respect to this, it is due to introduce how an IDS/IPS actually works internally. As well as the above presented classification based on where the IDS/IPS is installed, a further distinction can be done. In fact, based on the internal functioning of these devices, several types of IDS/IPS can be distinguished.

In particular, this distinction is based on how the system is able to recognize a threat [46]. First, signature-based IDS/IPS is able compare the observed network traffic to known malicious traffic signatures and patterns. In such a way, when the analyzed traffic profile matches the typical one of a known signature, that traffic is classified as anomalous and/or malicious, since it represents a potential threat. This kind of system is able to reach an high threat detection rate, generally limiting false positives, while the drawback is represented by the impossibility to detect unknown threats, since the detection is based only on known patterns.

On the other hand, a different approach is followed by the so called anomaly-based IDS/IPS. This second kind of system works by building a model of the "normal" behavior of the protected system. Consequently, all the analyzed traffic is compared to this model, and any deviation from the behavior considered as the traditional one is labeled as a potential threat. Therefore, this kind of approach introduces the possibility to detect also threats that the IDS/IPS has never seen before, while on the other hand the drawback is represented by the necessity of building a really accurate behavioral model of the protection scope, identifying well the "normal" behavior, otherwise the risk of false negatives, always dangerous, or false positives, especially dangerous in case of an IPS, is introduced. In addition, it is due to cite that, because of the lesser degree of specificity with respect to a signature-based IDS/IPS, which only focuses on the set of threats he knows, an anomaly-based IDS/IPS is generally less accurate.

Finally, it is due to mention how the actual detection of threats is performed by these systems. First, it is due to cite that the internal functioning of the system depends on the type od IDS/IPS that is considered. As specified by IBM[3], a signature-based IDS/IPS works by comparing observed network packets to specific signatures stored in a database it maintains. Consequently, if a packet triggers a match to one of the signatures, the correspondent mitigative action is performed, in case of IPS.

Differently, an anomaly-based IDS/IPS use artificial intelligence and machine learning to create and continually refine a behavioral model of normal network activity. The IDS/IPS compares observed network activity to the model and triggers corrective actions, in case of IPS, when it observes deviations from the behavior it considers as standard.

Before the introduction of machine learning in this context, the analysis was performed manually by a human analyst, which must search through vast amounts of data to find anomalies in the network traffic [47]. With respect to the intrusion detection problem, the usage of machine learning algorithms introduced significant improvements in the traffic analysis since they allow to completely delegate to a machine the extrapolation of threat-specific signatures, in case of a signature-based IDS/IPS, or typical characteristics of normal traffic, in case of an anomaly-based IDS/IPS, automating the characterization process and introducing the ability to detect threats which are potentially too sophisticated to be detected by a human being.

[3] https://www.ibm.com/topics/intrusion-prevention-system [Accessed: Aug 26, 2023]

5.2.1 IDS and IPS in the IoT world

Attack and anomaly detection in IoT infrastructures is a rising concern in the domain of IoT. With the increased use of IoT infrastructures in every domain, threats and attacks in these context are also growing commensurately. Numerous methods for improving data confidentiality, authentication, and access have been reported in the literature; however, even with these mechanisms, IoT networks are prone to multiple attacks aimed at disrupting the network. The growth, complexity, ubiquity, and diversity of the IoT expands the potential attack surface [48].

IoT devices communicate over networks, often using various protocols and communication mediums, which significantly expands the attack surface for potential attackers. In this context, provided the problems caused by IoT intrinsic insecure nature, introduced in Section 5.1, the presence of an efficient IPS solution is essential, in order to detect and prevent unauthorized access, attacks, and abnormal behaviors within a IoT networks environments. In addition, the lack of hardware resources that is usually found when considering these devices makes them particularly exposed to certain types of attacks, as will be presented in Section 5.4, which makes the presence of an efficient IPS solution crucial in an IoT network scenario.

Consequently, in light of these considerations and recalling the various existing IPS solutions presented Section 5.2, the most appropriate solution for this type of scenario appears to be the usage of a NIPS [49]. In fact, as aforementioned, this kind of solution is scalable and independent on the number of devices, and device types, falling within its scope of protection, therefore appearing to be particularly suitable for scenarios like IoT networks where the resources on board the single host preclude the installation of traditional host-based monitoring solutions.

In the context of IoT networks, considering a NIPS solution, it works by analyzing network traffic between IoT devices, gateways, and other connected components. It should be noted that, considering this kind of solution, the adaptability to IoT scenarios actually comes for free and, as introduced in Section 5.2, from the network topology point of view, an IPS must be placed inline with the data plane, right behind the firewall, in order to be crossed by the traffic entering and exiting the internal IoT network and potentially mitigate the detected threats.

In this context, it is due to mention that the IDS/IPS technologies oriented to IoT can be divided in two sub-categories: IoT-specific and IoT-agnostic [50]. An IoT-specific IDS/IPS solution is suitable in presence of devices using a particular communication technology and protocols, such as BLE (Bluetooth Low Energy)[4], designed for very low power operation and supporting multiple communication topologies, from point-to-point to broadcast. In this case, from a network topology point of view, the IDS/IPS must be necessarily deployed on the same local network of the device. The main focus of this type of systems is represented by control information of the specific technology and the predictions are based on control messages sent and received by IoT devices, essentially implementing a sort of protocol compliance check.

On the contrary, IoT-agnostic IDS/IPS do not depend on a particular IoT technology. This kind of IDS/IPS utilizes information available from the network traffic observation regardless of which technology is currently used by the devices, such as TCP/IP traffic. This class of IDS/IPS is suitable to be used also in an edge environment, out of the local IoT devices networks, since it can deal with traffic generated by heterogeneous devices leveraging different communication technologies.

Comparing these two types of IDS/IPS technologies, their characteristics can be evaluated in order to understand their effectiveness and suitability in different scenarios. Mainly, the advantage brought by an IoT-specific IDS/IPS over IoT-agnostic one, is the ability to detect low-level attacks, delivered using IoT-specific technologies and protocols, hence generated on the device-level. On the other hand, a single IoT-agnostic IDS/IPS is able to deal with a wider range of IoT devices, without the need to deploy an IoT-specific system for every communication technology object of monitoring.

Finally, it is due to mention the fact that, although the usage of a NIDS/NIPS appears to be, at least at first glance, the best choice for an IoT network scenario, given the repeatedly

[4] https://www.bluetooth.com/learn-about-bluetooth/tech-overview/ [Accessed: Aug 26, 2023]

mentioned characteristics of IoT devices, there are situations in which a host-based recognition and counteractive actions are preferable [49]. In these cases, traditional intrusion detection/prevention methods need to be modified and adapted for application to the IoT devices, taking into account their limitations, including constraints in the resources availability, limited memory and battery capacity, and specific protocol stacks used.

For this reason, in the literature, HIDS/HIPS solutions specifically oriented to the IoT world have been proposed [48]. In this context, always recalling the limiting characteristics of IoT devices, it is fundamental that the employed system was as lightweight as possible but still efficient. The term lightweight, in this context, does not refer to simplicity of the system, on the contrary, it is referred to the fact that the IDS/IPS should be able to perform its operations considering the limited available amount of resources in the devices.

Considering anomaly-based IDS/IPS, nowadays usually based on machine learning techniques, the aforementioned objective can be accomplished by simplifying or even avoiding the complex features extraction and selection steps, preferring an uncomplicated and limited number of features to be extracted from data. This simplification for using only a small subset of features introduces the advantage of reducing the system processing time, because of the lower time consumption of single, or a few, attributes acquisition from input data and the reduced time needed to extract a few features from a single attribute, with respect to the extraction of many features from the multiple attributes [48].

Intuitively, the drawback of this kind of technologies is represented by the lack in the ability to detect threats whose effect does not reflect on the selected features. In fact, by way of example, assuming a system relying on just two selected features, both related to the intensity of the traffic transiting on the monitored node with the purpose to detect DoS-like attacks, it will not be able at all to detect threats of a different nature not closely related to this type of change in the traffic profile, therefore protecting the device only under certain conditions.

5.2.2 IDS/IPS implementations

After having presented general aspects related to what an IDS/IPS is and how it works, it is due to mention some well-known implementations of this kind of technologies. In fact, as happens for the vast majority of technologies, also for IDS/IPS it is possible to find different products, which adopt different implementational choices, commercial and not. Considering the particular IDS/IPS scenario, depending on the different characteristics that an IDS/IPS may have, as presented in Section 5.2, each specific product may be more oriented to certain types of tasks (e.g., signature-based or anomaly-based detection) and ensure better performance in certain scenarios.

A famous IDS/IPS solution is indeed Snort[5]. Snort solution consists of an open-source framework which offers real-time intrusion detection and prevention capabilities. It primarily focuses on signature-based detection and offers the possibility of customizing the rule sets, also providing a rule-sharing system with other users. In addition, Snort also supports anomaly-based detection, and an optional IPS mode, which enables the software to take active mitigative actions.

Snort primarily operates at the transport layer of the ISO/OSI model. It operates by the sniffing the network traffic looking for potential security threats and breaches, employing a combination of signature-based detection and anomaly recognition mechanisms. Therefore, this product focuses on monitoring and analyzing the data exchanged between devices within a network, rather than diving into the application layer details.

Thus, To provide a practical example of Snort functioning, the following custom rule can be considered:

```
alert tcp any any -> $HOME_NET 22 (msg:"SSH Login Attempt";
    flow:to_server,established; content:"SSH-2.0"; nocase;
    sid:1000001;)
```

[5] https://www.snort.org/ [Accessed: Aug 31, 2023]

This rule aims to generate an alert when SSH login attempts, coming from any IP address and TCP port and directed to TCP port 22 of any host of the internal network, identified by the `$HOME_NET` Snort environment variable, are performed. In particular, Snort works by scanning the initial banner sent by SSH servers when a client connects and checking if it matches the one provided in the `content` clause of the rule, in this case the string "SSH-2.0", without considering case-sensitivity since `nocase` is specified. If a match is detected by Snort, it will generate an alert containing the message specified in the `msg` clause of the related rule, in this case the string "SSH Login Attempt". Lastly, a signature identifier (1 000 001) is specified by the rule in order to help rule management in Snort.

Another well-known product in this context is represented by Zeek[6]. This solution, formerly consists of a dynamic open-source framework capable of providing real-time intrusion detection capabilities. The main strong point of Zeek consists of its capability of extracting detailed information from various network protocols and generating logs and events that capture application-specific details.

Thus, Zeek can identify and report application-level behaviors, making it particularly effective at detecting complex threats and details that might be missed by solutions focusing exclusively on lower layers of the network stack. In fact, while Zeek provides some visibility into lower network layers, its main strength consists of the ability to extract rich application-layer information for threat detection, thanks to a series of protocol-specialized sub-modules which are part of its packet processing pipeline.

Zeek follows a passive monitoring approach, therefore it cannot be properly classified as an IPS, rather, it consists of a software, which can be installed on a wide variety of different monitoring targets, that quietly and unobtrusively observes network traffic, therefore implementing an IDS-typical approach.

Also for Zeek, an example rule, aimed at a better explanation of this technology and in particular for aspects about its capability of working at application level, can be provided:

```
@load protocols/ftp

event zeek_init() {
       FTP::enable_ftp(filenames=/.mp3$/);
}

event ftp_file(x: FTP::FileInfo) {
       print fmt("FTP file transfer detected: %s", x$filename);
}
```

In summary, this Zeek IDS rule focuses on FTP file transfers, specifically considering files with ".mp3" extensions. First, the specific protocol analyzer is asked to be loaded, via the `@load protocols/ftp` directive. Consequently, the second block of the rule indicates to enable the FTP protocol analyzer and specifies a filter condition through the `filenames=/.mp3$/` argument, in essence instructing Zeek to focus on FTP file transfers involving files ending with ".mp3". Finally, the last block of the rule defines an event named "ftp_file", which is triggered whenever an FTP file transfer is detected and prints a formatted message providing details about the specific file that triggered the event.

Another possibility for implementing an IDS/IPS is represented by the construction of a custom-developed platform. In particular this solution is the one adopted in the context of the present work, where an IPS has been developed as a module of the proposed IoT Proxy. The custom IPS is mainly based on two technologies: a custom-trained machine learning model and iptables[7].

iptables is a utility, from the command line, which can be installed on Linux systems and is configured by default on Ubuntu operating system up to version 20.04. It is a user space program

[6] https://zeek.org/ [Accessed: Aug 31, 2023]

[7] https://www.netfilter.org/projects/iptables/index.html [Accessed: Jun 04, 2023]

which has the purpose of configuring Linux set of NAT and packet filtering rules. iptables is actually a user space program used to interact with Netfilter[8]: the module of Linux kernel that is responsible for networking and firewalling.

iptables works at the network layer of the ISO/OSI model and it operates by the configuration and management of a set rules that control the flow of network traffic, providing the possibility to define the modalities in which packets are allowed, blocked, or modified as they traverse the system where iptables runs on. iptables is configured through a series of rule chains, each consisting of a set of rules that matches packet based on their attributes, such as source and destination IP addresses add/or ports, and protocols. When a packet is received, it is evaluated in order to understand the set of rules its attributes match, in a sequential way. Depending on the match criteria and specified action provided by the matched rules, the packet is accepted or dropped, or it can be allowed to be forwarded, or directed to further processing.

In order to further clarify the functioning of iptables, a practical example can be considered. Assume that the will was to block incoming SSH (Secure Shell) [51] traffic from the IP address 192.168.1.100. In this case, the iptables chain to consider is "INPUT", which comes into action when packets destined to the system iptables runs on arrive. "INPUT" is among the predefined chains provided by iptables together with "FORWARD", which contains rules for traffic that passes through the host and is destined to another party, and "OUTPUT", which rules the traffic that would be sent through the network by the host.

In order to block SSH traffic from 192.168.1.100, a rule must be added to the "INPUT" chain and that rule must match packets from that specific IP address and indicate to drop them. the rule in question is specified via the command line interface issuing the following command: `iptables -A INPUT -s 192.168.1.100 -p tcp --dport 22 -j DROP`. Analyzing the presented command, in order to understand iptables rules structure, it is worth remarking that, for this specific rule:

- `-A INPUT`: indicates that the rule has to be appended (`-A`) into the "INPUT" chain.
- `-s 192.168.1.100`: is the source IP address.
- `-p tcp`: indicates the protocol to consider is TCP.
- `--dport 22`: indicates the destination port is 22 (SSH).
- `-j DROP`: indicates that packets matching the rule should be dropped.

Now let's assume an SSH request from 192.168.1.100 arrived to the system. When the example SSH request packet is received by the system, iptables processes the rules in the INPUT chain and, since the action specified by the rule configured is "DROP", the packet will be discarded and not reach its intended destination, resulting in the block of the SSH connection attempt.

Therefore, as presented, iptables allows to decide the actions to be taken on the traffic that transit on the system through the configuration of a set of rules and can actively affect network traffic. Therefore, assuming the possibility of dynamically configuring iptables rules, it turns out to be a particularly suitable solution to perform the active mitigative actions typical of an IPS.

On the other hand, for the aspects related to the detection task, an especially appropriate solution can be identified in the usage of a machine learning model, as it is done in the context of the present work. In this scenario, both supervised and unsupervised machine learning techniques, presented in Chapter 4, are suitable for the purpose, since they can be the base to implement, respectively, a signature-based and an anomaly-based IDS/IPS.

Once trained, the model can effectively recognize threat signatures or identify deviations from standard network behavior, assuming a good accuracy provided by the model. This predictive capability enables the machine learning model to detect threats and attacks, and sophisticated

[8] https://www.netfilter.org/ [Accessed: Jul 15, 2023]

intrusion attempts that traditional rule-based methods might miss. Thus, considering the detection capabilities provided by a machine learning model and the ability of iptables to actively take actions on network traffic in transit on the system, an IPS solution can arise from the combination of these two technologies.

In the context of the present work, the decision to adopt a custom-developed platform incorporating iptables and a custom-trained machine learning model for the IDS/IPS implementation was driven by its unique strengths. This solution offers a high level of customization, enabling the integration of tailored rules and a machine learning model trained specifically for the analyzed IoT network environment, able to automatically learned and derive signatures through the observation of labeled data, which represents an additional significant advantage. Unlike ready-to-go solutions like Zeek, which rely on predefined signatures or behavioral patterns, this custom approach is more adaptable to context-specific threats and, as it will be presented in Chapter 6, the usage of a custom deployed product allows to integrate innovative features such as oblivious authentication. This solution capitalizes on the advantages of both machine learning and finely-tuned rule-based detection, making it a strategic choice in the context of the present work, seeking a precise and adaptable security solution.

5.3 VPN gateways

In the ever-evolving landscape of digital connectivity, network security, as introduced in Section 5.1, stands as a paramount concern. The intricate web of interconnected devices and systems that constitutes the modern digital world is rife with potential vulnerabilities, making the safeguard of, potentially sensitive and/or critical, data and communications a top priority. Among the array of tools and strategies at the disposal of cybersecurity professionals, the VPN emerge as one of the most pivotal solutions for ensuring secure and private communications over the Internet.

In order to understand what a VPN is, one could start with the semantic analysis of the acronym [52]. First, a VPN is virtual, which means that the VPN is built upon an existing, physical and potentially shared network infrastructure. The virtual topology and the physical network infrastructure are usually administered by different entities. To provide a practical example, a given company can build its own VPN over a network provider physical infrastructure and, in this scenario, the VPN is fully administered by the company, while the physical network maintenance and management is up on the provider. Therefore, a VPN can be defined as a communication environment built by controlled segmentation of a shared communications infrastructure to emulate the characteristics of a private network.

In addition, a VPN is private, which indicates that it reflects all the typical features of a private network, for example, the possibility of restricted access only to certain users and entities, which are provided with the ability to interact with resources and services on the network. Moreover, traffic originating and destined within the private network only crosses nodes belonging to the private network, which means that the traffic of a private network is not affected or affected by other network traffic outside the private network.

Finally, a VPN is a network, which means that it provides connectivity in order to exchange information to entities connected to it. Therefore, a VPN consists of two or more entities connected for the purpose of transmitting, exchanging or sharing data and resources.

Summarizing, a VPN is a powerful tool that enhances online privacy, security, and accessibility by establishing secure and encrypted connections. This connections mask the internal IP addresses and encrypt data traffic, effectively creating a private "tunnel" through which information can travel safely across a shared network infrastructure, potentially the public Internet. A VPN offers a wide range of benefits, from safeguarding sensitive information to bypassing geographic content restrictions. There are several types of VPNs, each designed to meet specific needs and use cases.

One of the main need for the adoption of a VPN comes from the perpetual organizations growth [53]. It is not rare for modern companies and organizations to have multiple offices scattered across different geographic regions. In this context, it is crucial for companies to guarantee connectivity among data and resources, introducing the need for a cohesive and secure private

network infrastructure. Furthermore, the rise of remote work, which has been accelerated by recent global events, underscores the necessity by the remote workers to access the private company network from out of the traditional office, mainly in order to interact with the company private resources, making the secure remote access to the company internal resources a paramount aspect.

Traditionally, private networks provide connectivity among various interested entities through a set of links, often consisting of dedicated circuits. In this modern scenario, maintaining a cohesive and secure network infrastructure becomes a complex challenge for companies, because of the difficulty, or even often impossibility, to guarantee physical network presence in many different geographic areas, potentially all over the world.

In this context, the VPN emerge as a perfectly suitable solution to address this challenge. In fact, although several technologies exist in order to provide secure and efficient connectivity among geographically distributed branch workplaces, strategic partners, and remote workers, most recently Internet-based VPNs have evolved as the most secure and cost-effective mean of achieving these goals [53].

Technically speaking, a VPN general functioning provides the creation of a private and encrypted channel over a public network, such as the Internet. This encryption safeguards sensitive company data from potential breaches and cyber-threats, such as unauthorized accesses to company resources, regardless of the physical location of employees or company sites.

Based on the virtual network topology created and on the entity of the actors involved in the VPN establishing, a first classification of several possible types of VPNs can be made and each VPN type is designed to satisfy different requirements and use cases. First, it should be noted that a VPN connection establishing always provides the presence of at least one VPN gateway, also known as VPN terminator. Usually, the role of gateway is played by a firewall, but actually this role can be played by any host within a network.

Based on the nature of the connection that is designed between the VPN gateway an the other party, and based on the nature of the other party as well, several types of VPNs fundamentally be distinguished [54]. Taking Figure 5.2 as reference, gateways are represented by the item labeled as “GW”, while the items with “IH” label represent hosts of private networks, the one labeled with “RW” represents a road warrior and “RS” consists of a remote server.

When designing a VPN, the following types of connections are possible:

- site-to-site VPN: this type of VPN permits to securely link the local network of one site, such as a company workplace, to the local network of another site, creating a connection between two VPN gateways. Again, the connection is established on a non-private channel, such as the public Internet network, between two VPN gateways topologically located at the border of the two private network to connect. By creating a secure connection above this insecure channel, hosts belonging to the two linked local networks can be communicated as if they were in fact on the same local network. Taking Figure 5.2 as reference, thanks to the red colored connection, internal host “IH 3” is able to communicate with internal hosts “IH 2” and “IH 3”, as they all belong to the same local network.

- remote access VPN: this kind of VPN allows a remote client to securely connect to a private network, directly establishing a tunnel between itself and a VPN gateway placed on the other side. A secure link is established, through a public network, usually Internet, between the remote client and the VPN gateway on the local network, so that the client can access all its services and resources, such as files and internal corporate portals, from a geographically different location. Taking Figure 5.2 as reference, thanks to the connection represented in blue, road warrior “RW 1” is able to communicate with internal hosts “IH 1” and “IH 2” as they were on the same local network;

- host-to-host VPN: this third type of VPN can be seen as a particular case of a site-to-site VPN, where one of the two VPN gateways involved in the creation and the establishing of the secure link is directly the resource to be reached by the parties from the other side. To provide a practical example, an host-to-host VPN connection can be established between a company firewall, placed at the border of a corporate local network, and a remote single

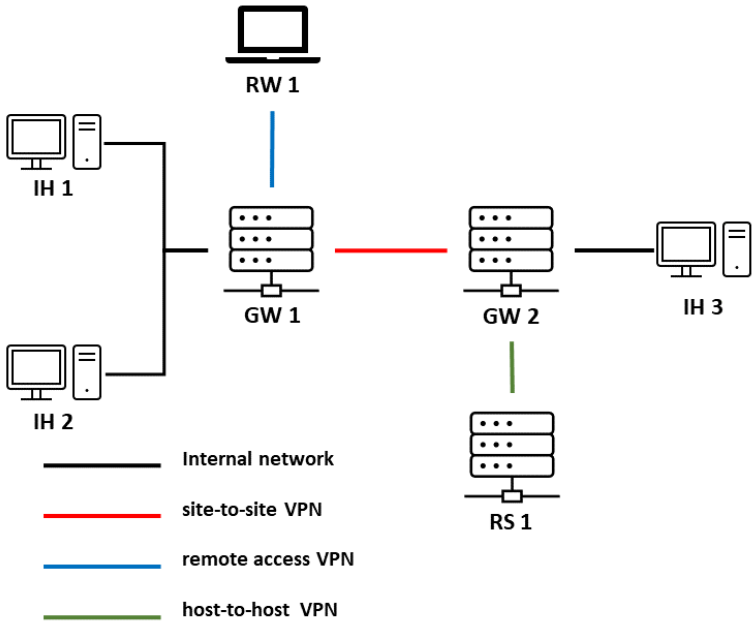

Figure 5.2. Possible VPN types

server, so that the hosts of the local company network can communicate with the remote server as they were on the same local network. Taking Figure 5.2 as reference, through the connection depicted in green, internal host "IH 3" is able to communicate with remote server "RS 1" as they were on the same local network;

After having presented the first classification about VPN types, based on the actors involved in the creation of the tunnel, a second classification, related to the various architectures for a VPN implementation, has to be discussed. Typically, VPNs are implemented either at the network layer or at the link layer [52]. However, since link layer VPNs come into play when the underlying shared infrastructure is based on a circuit switching link-layer technology, which represents a quite rare situation in the nowadays technological scenario in which the vast majority of networks are based on packet switching, they have lost almost all of their popularity and are only used in particular or legacy scenarios.

Actually, nowadays, also application layer VPN implementations can be found. This kind of VPNs are primarily designed for remote access scenarios, allowing individual users or systems to securely connect to specific applications or services hosted on a network. This approach can offer some advantages, such as control over access permissions, enhanced security for specific services, and ease of use for remote users. Application layer VPNs often utilize TLS [55] as the underlying protocol for encryption, making them accessible via a standard web browser, which simplifies client-side setup and accessibility.

On the other hand, the most common type of VPNs nowadays is indeed the network layer VPN. A network layer VPN is traditionally based on IP protocol at the network layer and, for this kind of VPNs, two different actual implementations are possible. The first alternative is represented by the implementation of a network layer VPN by tunneling, which consists in a technique that provides the creation of a tunnel which connects two points of a VPN across a shared network infrastructure, such as Internet. When working in tunneling mode, the end-points of the tunnel, which play the role of VPN gateways, are common nodes of the VPN and of the shared network infrastructure and, from the architectural point of view, the VPN is a collection of tunnels established over the shared network infrastructure.

When a network packet leaves one end of the tunnel it is enriched with an additional IP header, whose destination corresponds to the other end of the tunnel. Such packet is then routed across the shared network infrastructure to the other end of the tunnel, based on the modified IP destination, and, when it reaches the latter, the additional IP header is stripped away. In this way, the original packet is re-created and it is routed to the initial destination across the remote local private network.

The main advantage of tunneling is represented by the flexibility it provides. In fact, assuming IP as the network-layer protocol within the shared infrastructure, thanks to this sort of encapsulation, the original network packets could be based on any Layer 3 protocol, not only IP, and use a completely different routing protocol with respect to the shared network infrastructure they are routed across. Still, they remain able to be IP-routed across the shared infrastructure to the remote VPN gateway.

The major concern when considering tunneling is represented by the security. In fact, since this basic technique does not natively provide any protection elements, the packets belonging to the private communication travel unencrypted over the shared network infrastructure, which exposes them to be easily eavesdropped and read by potential attackers. Thus, the tunneling technique alone is not sufficient to build a secure VPN.

The solution in this context consists of the aforementioned network layer encryption, which provides a secure mechanism for the implementation of a network layer VPN. The standard solution in this scenario, proposed and standardized by IETF (Internet Engineering Task Force), is represented by IPSec [56]. IPSec is designed as a collection of protocols, authentication and encryption mechanisms, which aims to guarantee interoperable, high quality, cryptography-based security at the IP layer, offering protection for IP and contextually to upper layer protocols.

An IPSec packet contains an IP header and it can be routed accordingly to it by traditional IP routers from one VPN gateway to another. Through the usage of encryption algorithms and a set of encryption keys, negotiated and exchanged between two VPN gateways, along with the algorithm parameters, using the IKE (Internet Key Exchange) protocol (which is part of IPSec specifications), the original IP packet is securely encapsulated, creating an IPSec packet. Moreover, the IPSec packet can also be equipped with an authentication header, which which serves to ensure the validity of the entire IPSec packet, so that the packet receiver could be able to check if the packet has modified while in transit across the shared network infrastructure.

In the context of IPsec, the security functionalities are offered by two integral protocols which are able to provide distinct but complementary security functions, namely AH (Authentication Header) and ESP (Encapsulating Security Payload). Each protocol can be used as standalone in order to protect an IP packet, or both protocols can be applied together to the same IP packet.

The AH focuses only on authentication and integrity and it is able to provide data integrity, data origin authentication, and optional replay protection. AH protocol ensures data integrity through a message digest that is generated by a cryptographic algorithm, such as HMAC-SHA, and appended at the end of the packet. Data origin authentication is ensured by the usage of a shared secret key, exchanged during a previous phase, to create the message digest. Finally, replay protection is provided by using a sequence number field within the AH header. AH ensures authentication and integrity of the whole IP packet, with the exception of certain header fields that are legitimated to change while in transit, such as the Time To Live (TTL) field.

On the other hand, The ESP protocol allows different modes, providing only confidentiality, only authentication, or both confidentiality and authentication to the payload of the IP packet. If ESP is configured to provide authentication functionalities, it provides the same functionalities and uses the same algorithms provided by AH, but the coverage is different. AH authenticates the entire IP packet, including the outer IP header, while the ESP authentication mechanism authenticates only the IP payload, not providing any kind of protection to the outer IP header.

Lastly, it is due to introduce some aspects about VPN performance. Unsurprisingly, the features offered by the usage of a VPN do not come for free, in particular in terms of network throughput. In order to discuss these issues, for the discussion and evaluation of the performance and overhead introduced by using a VPN, the functioning of OSLVs (Open-Source Linux-Based VPN) implementation can be taken as a reference [53].

Considering this particular VPN implementation, the data path followed by a packet in correspondence of the VPN gateway encountered on its route from one private network to another VPN side, is essentially divided into three parts. First, the packet is received by the VPN terminator on one of its network interfaces and it is handled by the IP routing daemon, a daemon process implementing routing protocols in Linux systems which decides the packet next-hop and/or egress interface by using the longest prefix matching technique on its destination address. Secondly, since the IP routing daemon recognizes that the destination address belongs to a private network for which it has no direct routes on any network interfaces, this packet will be automatically dropped unless the routing table is updated to route it through a VNI (Virtual Networking Interface). A VNI is an abstract virtual device, specifically created during the VPN initialization phase to permit the exchange of packets between the IP routing daemon and the VPN daemon, which consists of a daemon process responsible for VPN packets handling. Consequently, The VPN daemon compress and applies other cryptographic operations to the packet, which, after the other side VPN terminator public IP address and port have been determined, is enriched with additional IP headers/trailers, in order to use tunneling technique, and is sent it as a normal IP packet over the shared network infrastructure. Finally, at the receiver side, the exact reverse steps are performed and the original packet is retrieved.

The aforementioned relatively simple software architecture, because of the additional computational steps to be taken by a VPN packet, intuitively introduces network performance degradation elements. The first reason is due to the spacial overhead caused by the introduction of additional headers to the original IP packet, which represent the main cause, and by the cryptographic functions application. This enlargement of the original packet size introduces a deterioration in terms of bandwidth occupation, which is the bandwidth portion used to transport actual information. Ultimately, employing a VPN results in an increase in latency. This is mainly attributed to the supplementary stages that data packets need to traverse and the resource-intensive operations proper of cryptographic functions.

In conclusion, while the utilization of a VPN may lead to a degree of network performance degradation, it remains an invaluable solution. The added layers of security and the ability to overcome geographical restrictions far exceed the introduced latency increase and data processing complexities, making the VPN a truly indispensable tool nowadays.

5.3.1 VPN and IoT

In the realm of the Internet of Things (IoT), where a multitude of devices are interconnected to collect and exchange data, security becomes an even more critical concern. As repeatedly stressed in this debate, IoT devices often possess limited computational power, making resource-intensive security protocols impractical. Therefore, taking into consideration the aforementioned various type of VPN solutions, the site-to-site VPN logically appears to be a suitable solution in this context.

A practical example in this context can be represented by two geographically separated networks, consisting of an IoT devices network, where all the IoT devices in charge of collecting and sending data are placed, and a centralized management network, where an hypothetical central management platform in charge of collecting these data is placed. As mentioned in Section 5.3, the usage of a site-to-site VPN allows to establish a secure channel between two, even geographically distant, local networks, thus representing a really suitable solution in this context.

Deploying a site-to-site VPN in such scenarios can significantly enhance the efficiency and security of IoT devices networks. In fact, in case this solution is chosen for the purpose, all the VPN-related operations, and consequently all the computational load introduced by them, is demanded to a VPN gateway. Therefore, providing two systems, one for each of the aforementioned networks, to act as VPN terminators connected each other, this configuration allows IoT devices to communicate securely with the central system without directly bearing the computational burden of encryption and decryption.

The key advantage of offloading these resource-intensive tasks, such as encryption/decryption, data compression, or complex authentication procedures, onto a VPN gateway lies in its capacity

to handle these operations efficiently. In fact, the systems working as VPN gateways are usually equipped with an adequate amount of computational resources and may also have in place some sort of dedicated cryptographic accelerators, making them suitable to perform computationally complex tasks without influencing IoT devices.

In this way, this solution not only ensures the security and privacy of IoT communications, offering the typical feature of a VPN, but these security features are provided without burdening in any way on IoT devices, instead offloading the resource-intensive tasks from these devices totally demanding them to a more powerful device.

Following this approach, on one hand preventing the execution of computationally heavy tasks on the resource-constrained IoT devices and on the other hand demanding these tasks to a more powerful device, depending on how well the gateway VPN is equipped, it is possible to maintain encryption and data security at a high standard, introducing a benefit for both security, implementing complex protection mechanisms, and for performance, allowing IoT devices to operate more efficiently and focusing on their specific purpose.

Finally, it's important to note that the best VPN solution to choose when considering an IoT scenario can also depend on the use case and specific requirements. While the adoption of a site-to-site VPN solution, as aforementioned, offers several advantages, other VPN types like a remote access VPN or a host-to-host VPN might be more suitable for certain situations, such as respectively, by a way of example, a single IoT device that needs to be connected to the central management platform placed on the cloud or individual geographically distant IoT devices that need to interact each other in a distributed architecture [57].

Obviously, in light of the considerations repeatedly mentioned about the computational capacity and resources limitation typical of IoT devices, considering the implementation of a remote access or host-to-host VPN solution, therefore involving the introduction of additional computational load on the IoT device, there is a need to adopt lightweight solutions. In this context, it is due to mention the lack of standardized technologies following a lightweight approach, essentially using lightweight cryptography algorithms. However, the US NIST (National Institute of Standards and Technology) is taking steps towards an attempt at standardization in this context and, in early 2023, selected the Ascon[9] cryptographic algorithms family as the solution to be chosen for the future standardization of lightweight cryptography.

In conclusion, it is worth remarking that the most suitable choice should always be guided by a careful consideration of factors like device capabilities, security assurances required, and the global structure of the architecture of the IoT network. Taking these factors into account, in order to impact as little as possible the few resources available and do not significantly affect the operation of the IoT device itself, robust and secure communication channels for their IoT devices can be established, ensuring the successful integration of IoT technologies in modern environments.

5.3.2 VPN terminators implementations

After having discussed general aspects about VPNs, explaining what they are and how they work in general, it is worth to introduce some well-known available implementations. In fact, when deploying a VPN, several products, commercial and not, are available and, depending on the specific scenario, a specific solution can be the most suitable with respect to the others, reasoning in terms of costs and needed performance and security assurance.

First, it is due to mention that, when considering VPN solutions, commercial solutions, especially when considering a business scenario where the business continuity represents a crucial factor, are widely adopted. Commercial solutions for VPN provided by vendors like Cisco and Fortinet, well-established vendors in the network security industry, offer a higher level of reliability, performance and support. These ready-to-go solutions offer advanced features such as integrated

[9] https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon [Accessed: Aug 31, 2023]

intrusion detection/prevention capabilities, endpoint security integration, and extensive reporting, which can help organizations, especially those without the availability of professionals in the security field, protecting their networks. Furthermore, commercial vendors usually invest heavily in research and development, ensuring that their products remain up-to-date with the latest security threats and regulatory requirements.

Considering instead non-commercial and open source products available, several solutions can be adopted when designing a VPN. One of these is indeed represented by OpenVPN[10]. This solution supports all the VPN types presented in Section 5.3 and is highly versatile, offering cross-platform compatibility and flexibility in configuration, which makes it a suitable solution for a wide range of use cases.

OpenVPN is based on a client-server architecture and it operates at application layer by utilizing the TLS protocol for secure key exchange and encryption. When a client initiates a connection to an OpenVPN server, both parties start a process of mutual authentication and key exchange, similar to the TLS handshake phase when considering web browsing. In this phase, client and server exchange certificates to verify their identities each other. Once authenticated, encryption parameters are negotiated, including the choice of encryption algorithms and keys. This encryption parameters are then used to encipher packets sent between the client and server, rendering them protected to potential eavesdroppers.

When considering open-source solutions, in particular in relation to Linux environment, it is due to introduce xfrm[11]. xfrm consists of an IP framework which offers features in order to transform IP packets, making it possible to perform operations such as payload encryption. The features offered by xfrm are used to implement the IPsec protocol suite, presented in Section 5.3, in Linux systems.

Therefore, considering the features it offers, xfrm can indeed be used to build a full-fledged VPN solution. However, working with pure xfrm typically requires a deep understanding of networking concepts, security protocols, and low-level configuration skills, since it is deeply embedded in the Linux kernel's networking stack and operates at a lower level. While this level of control allows to build fine-tuned security configurations, it also introduce the need for a high level of expertise and attention to detail to ensure correct functioning, requiring the practitioners to manage, and sometimes also define, packets transformations, encryption algorithms and authentication mechanisms manually.

Instead, always considering network layer VPN based on IPsec, higher-level solutions, that offer user-friendly interfaces and automatically abstract much of the complexity in the low level interactions, are available. One of these solutions, particularly relevant in the context of the present work, is represented by strongSwan.

strongSwan is an open-source, modular and portable VPN solution. As aforementioned, it is based on the IPsec protocol suite, used to provide privacy and authentication services at the IP layer, therefore securing network communications through the addition of confidentiality, integrity, and authentication control functionalities to IP packets.

strongSwan supports all the VPN possible types explained in Section 5.3 and provides robust encryption, authentication, and key exchange mechanisms, making it a valid choice for the creation of secure VPNs. With strongSwan, which leverages the IKEv2 protocol to negotiate cryptographic parameters, generate session keys, and authenticate the involved actors, VPN connections can be established using various strong and modern encryption algorithms, coupled with strong authentication mechanisms.

strongSwan enhances a plain IPsec implementation in several significant ways. First, it provides user-friendly configuration, which means that while plain IPsec often requires manual configuration of numerous parameters, strongSwan simplifies this process providing the possibility to configure the IPsec VPN through configuration files and command-line utilities, making the overall configuration process easier for practitioners.

[10] https://openvpn.net/ [Accessed: Sep 02, 2023]

[11] https://man7.org/linux/man-pages/man8/ip-xfrm.8.html [Accessed: Sep 02, 2023]

Starting from a raw installation of strongSwan, some steps are required to setup the VPN gateway. First, the key and the certificate of a certification authority, used for the VPN gateway authentication, must be created or imported. The certification authority certificate will then be shared with each client that will need to connect to the present one. Suddenly, the same key and certificate generation must be performed for the gateway itself. Moreover, strongSwan is designed following a modular approach and it allows practitioners to add or remove components at need, enabling also the implementation of additional features or customizations. Finally, as briefly introduced above, another important feature introduced by strongSwan with respect to what plain IPsec offers is represented by the possibility to use strong authentication mechanisms, such as PKI[12] or pre-shared keys, as well as providing native integration with RADIUS [58], thus providing more extensive options for authenticating VPN users and devices.

In the context of the present work, strongSwan represents the chosen VPN implementational solution. This choice is based on considerations about strongSwan flexibility and inherent compatibility with Docker platform, which makes it perfectly compatible with the followed lightweight virtualization strategies. The features offered by strongSwan in terms of security, flexibility, and deployment adaptability, as well as the presence of a user-friendly interface provided by the platform to ease the configuration process, make it a suitable choice for VPN implementation in the context of the present work.

5.4 Mitigated attacks

In the context of an always increasingly interconnected world, the need for strong cybersecurity measures arise, in order to protect people, organizations and systems from cyber-threats. Considering the exponential growth of digital interactions and the critical role of networks, it becomes evident that network security field, in particular, represents a fulcrum in the defense against cyber-threats.

In fact, in nowadays digital landscape, the network, often Internet, represents the distribution channel for a vast amount of data, applications, and services, making it become also an attractive mean for attackers, which can exploit this widespread connectivity to deliver a multitude of threats which make use of the network as their attack vector.

The range of cyber-threats delivered using the network as the attack vector encompasses a diverse and constantly evolving variety of malicious activities. These kind of threats leverage the pervasive connectivity of the digital landscape to infiltrate, disrupt, or compromise targets, represented by systems, data, and/or services. Examples among the most prevalent network-borne cyber-threats can be represented by DoS (Denial of Service) attacks, which inundate targets with traffic in order to make them interrupt the provision of the service to legitimate users, or malware propagation, capable of infiltrating networks and causing malicious consequences such as data breaches.

Considering the particular IoT scenario, which introduces all the repeatedly underlined implications such as the low amount of resources available or the lack of standardization in this area, some of these threats and attacks are particularly effective. In this context, VPNs and IDS/IPS solutions, respectively presented in Section 5.3 and Section 5.2, are two of the most effective countermeasures to be implemented in order to mitigate threats.

In general when considering network-delivered threats, one of the particular signs from which the existence of a threat ca be suggested is the presence of anomalous traffic within the network. This aspect is discussed in Section 5.4.2. In the context of IoT, one of the major security problems is related to traffic eavesdropping, presented in Section 5.4.1. Moreover, an additional major security concern in this context is represented by DoS attacks, where the attacker floods the network with a large volume of data to prevent nodes from using the services. One particular type of DoS attack is presented in Section 5.4.3. Moreover, when considering the particular IoT

[12]Public Key Infrastructure: is defined by all the policies, procedures, hardware and software involved in the creation, managing, distribution, usage, storage and revocation of digital certificates.

environment, several specialized threats in the compromise of this type of device exist, as it will be discussed in Section 5.4.4 which presents one particular instance of IoT-oriented attack.

5.4.1 Traffic eavesdropping

In the realm of IoT, one significant security concern is the exposition of communication channels to eavesdropping. In fact, often, IoT devices do not use any form of cryptographic techniques in order to ensure data confidentiality and their communications take place totally in clear [59]. This lack of any form of traffic encryption imply that data transmitted between these devices and central systems or between devices each other can be intercepted and read by a potential malicious actors with relative ease, with the only requirement to have access to the communication channel, thus, this scenario becomes even more problematic if wireless networks are considered.

Moreover, the absence of encryption or any form of confidentiality measures which, as said, exposes data exchanged during the network communications to eavesdropping, becomes an even more critical point if considering that, as introduced Chapter 1, IoT devices could communicate also sensitive data through the network, such as user personal information or industrial control signals, potentially compromising privacy, proprietary data, and even the security of IoT ecosystems.

In addition, it should be noted that the complete lack of encryption mechanisms does not only compromise the IoT device security but also represents a serious risk for the local network to which the IoT device is connected. In fact, by way of example, it could happen that in order to access a Wi-Fi network, the IoT device must send the Wi-Fi network identifier and the password through the network, as part of the process to access the network. Therefore, in this case, an eventual attacker would have the possibility to retrieve the Wi-Fi password reading it unencrypted by simply sniffing the communication channel, introducing for the attacker the possibility to sniff all further data and perform several malicious activities from within the network.

One suitable solution in order to mitigate the problem of traffic eavesdropping in the IoT context is the utilization of a VPN, which represents the adopted solution also context of the present work. A VPN, establishing a secure encrypted tunnel for data transmission, makes intercepted data unreadable to eavesdroppers so that allowing to IoT devices to ensure encryption of their communications, independently of the underlying network infrastructure or the security measures provided or not by the device itself.

In conclusion, the issue of traffic eavesdropping in IoT networks, often due by the transit of plain text data related to IoT devices communications within the channel used, represents a serious risk for the IoT device itself and for the whole surrounding network. In this scenario, the usage of a VPN represents a suitable solution, offering security functionalities to IoT devices, in particular traffic encryption, in order to avoid the access to potentially sensitive data by eventual attackers.

5.4.2 Device authentication failure detection

Sometimes, the presence of anomalies within a network can manifest in the form of repeated authentication failures. This pattern occurs when a user or entity repeatedly attempts to gain access to a network, a system or a service using incorrect or unauthorized credentials. The monitoring of these repeated authentication failures can represent a crucial task for security, as it can serve as an early warning system for potential security breaches.

Traditionally, authentication is based on the demonstration of knowledge and possession of some kind of secret, such as a password or a key. A practical example in this context can be represented by the authentication process in place when trying to access to a web service or a private network, which usually requires to provide a username and password. These traditional kinds of authentication processes are handled at application level and therefore also the countermeasures, which usually consist in the block of the account for which repeated failed authentication attempts were made, are implemented at application level.

The scenario differs from the traditional one when switching instead to an oblivious authentication paradigm, a particularly relevant topic in the context of the present work. This approach aims to build an authentication mechanism based on the categorization of parties according to their characteristics, represented by IoT devices device type and/or category in the context of the present work, through the analysis of statistics calculated over the network traffic related to such devices, making use of machine learning techniques.

Considering the particular IoT scenario, in which the devices are generally special-purpose and show a well defined and almost linear traffic profile, this approach turns out to be particularly suitable, since it is feasible to detect whether the characteristics of network traffic related to a given IoT device are in line with the typical traffic profile expected for it or if they deviates from it.

Therefore, switching to this paradigm, the network traffic related to IoT devices is continuously monitored and the device authentication becomes a perpetual process which constantly checks if the traffic profile related to the specific device respects the expected one for it or not. Thus, in this context, the concept of failed authentication radically changes in meaning and it can be exactly associated to the concept of abnormal traffic, which means these two concepts coincide.

To provide a practical example, a temperature sensor can be taken as reference device. This device likely shows a well-defined traffic profile, for example showing periodic impulses of network traffic in which the temperature is communicated to another entity such as a central system. If the traffic profile related to this device deviates significantly, for example by showing a continuous flow of data, perhaps also of a different nature from those expected, the device authentication is considered invalid.

The proposed IDS/IPS solution in the context of this work, that will be presented in Chapter 6, practically implements also the oblivious authentication paradigm, enriching the set of features offered by a plain IDS/IPS and providing also this possibility to authenticate monitored targets through the analysis of their traffic profiles.

5.4.3 TCP Flooding

As introduced in Section 5.4, in the context of IoT devices networks, one of the major security concerns is represented by DoS attacks. Again, the reason stands in the lack in the amount of resources usually available in IoT devices, which makes them easily saturable and consequently particularly prune to this kind of attacks [48].

A DoS is a kind of cyber-attack which aims to prevent targets to provide their service as they would in a normal way, realized through the flooding of the network with a large volume of data in order to saturate the available resources. Several types of DoS attacks can be distinguished and the categorization is mainly based on the chosen protocol to deliver the attack. One particular way of delivering a DoS attack is to flood the target using TCP [60] protocol, thus realizing a TCP flooding attack.

TCP is a core communication protocol in computer networks and allows the reliable and ordered data transmission between devices. This protocol operates using a connection-oriented approach, where a client and a server establish a connection before the actual data exchange begins. This connection setup is realized via a process called three-way handshake, represented in Figure 5.3.

Essentially, the client sends a TCP packet, with the SYN (Synchronize) flag set, to the server, indicating its will to start a connection. Upon the reception of this SYN packet, the server responds with a TCP packet with both the SYN and ACK (Acknowledgment) flags set, to assure the client that the previous packet has been received. Finally, the client sends an ACK packet back to the server, confirming the acknowledgment. From that moment on the client and the server are synchronized and ready to exchange data.

The behavior of the TCP three-way handshake can be exploited by the attackers to perform a TCP SYN flooding attack in the network. In particular, this kind of DoS attack exploits exploits the resources allocated for managing these connections. In fact, the exact cause that makes this

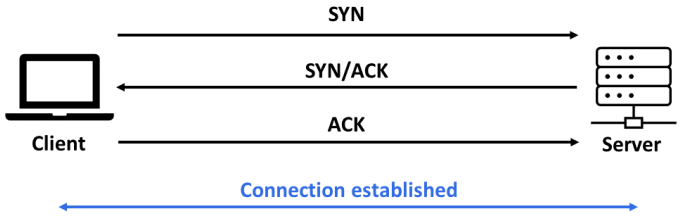

Figure 5.3. TCP three way handshake

attack possible stands in the way in which the server manages connection requests. When the server receives a TCP SYN packet, an entry is created in its flow table, which is the data structure used to manage TCP connections and store receipt of the SYN packets for each connection. A TCP SYN flooding attack aims to exhaust the table entries by sending multiple connection requests to the target and contextually blocking the response ACK once the SYN/ACK packet is received. This situation takes the name of half open connection and it causes heavy usage of resources for the target, resulting in the impossibility of the legitimate users or client to connect to the target server.

In addition to the saturation of the victim flow table, the TCP SYN flooding attack has an additional effect, which is due to the reliable nature of TCP protocol. When the server realizes, via a kind of timer, that the ACK packet for a specific connection has not been sent from the client, it continues to send SYN/ACK packets at each timeout, thus causing a waste of resources in terms of network card usage. The network scenario in case of an ongoing TCP SYN flooding attack is represented in Figure 5.4.

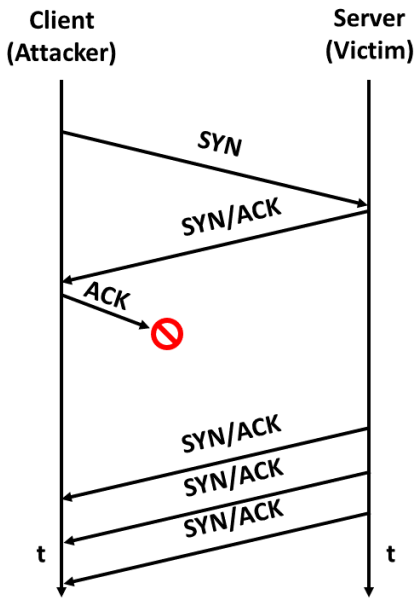

Figure 5.4. TCP SYN flooding attack

In this context, a possible countermeasure can consists in the blocking of the source IP address of the attacker. In fact, it is unlikely that a legitimate client needs such a large amount of different TCP connection to the same server, therefore it could be possible to limit the number of connection requests coming from a client to a non damaging amount. This mitigative action is however not effective, since it can be trivially bypassed because of the possibility of IP address spoofing by the attacker, which consequently allows to modify the source of the requests, making the server perceive that each request comes from a different client, thus bypassing this restriction.

Moreover, apart from being particularly sensitive to DoS, given the large amount of IoT devices in circulation and the usual lack of protections for them, IoT devices show an additional affinity with this type of attack, as they can also become the mean for the generation of the big amount of traffic needed for DoS, representing attractive victims for potential attackers who aim to create a botnet [61].

In fact, obviously, in order to achieve the desired effect via a DoS attack, a crucial requirement for attackers is the availability of resources capable of generating a large amount of traffic. Therefore, a botnet, typically composed by malware-infected devices known as bots which operate under the command of a single malicious controller, presents itself as an ideal weapon for attackers in this context.

The usage of compromised IoT devices in order to launch DoS attacks proved to be a winning technique, testified by the effectiveness of the attacks launched by IoT devices infected with Mirai malware: a potent malicious software capable of transforming IoT devices, among the others, into remotely manipulated bots, effectively making them become part of a botnet. The IoT devices infected with Mirai were used in 2016 to perform several large-scale attacks, reaching peaks of 1 Tbps (Terabit per second) of generated traffic [61].

In conclusion, while reiterating the particular affinity of IoT devices with DoS attacks, the importance of protecting such devices from this type of attacks is again underlined. The IDS/IPS module proposed in the context of this work, discussed in Chapter 6, is capable of recognizing and actively blocking an ongoing TCP SYN flooding DoS attack, thus providing a suitable and proactive security solution in this scenario.

5.4.4 RTSP bruteforce

The last type of attack presented consists in a IoT-specialized threat, namely RTSP bruteforce. Before providing the details about the functioning of this attack, it is due to introduce what RTSP is and how it works. RTSP (Real Time Streaming Protocol) [62] is an application-level protocol for control over the transmission of data, particularly video and audio, in real-time.

RTSP, the standard protocol in this context, provides the establishment of a client-server connection in order to allow the control of multimedia content streaming sessions, providing several commands useful in order to control or navigate the stream. RTSP functioning is quite straightforward. Essentially, an initial request is issued by the client, usually represented by a web browser or in general a system able to display multimedia content, specifying the desired media content and some parameters. Consequently, the server, which usually consists of an IoT camera, responds by acknowledging the request and transmitting the requested media content.

The main concern about RTSP stands in the lack of security provided by the protocol. In fact, as specified in RFC-2326, the reference one for this protocol, RTSP shows many similarities in terms of usage and syntax with HTTP, and also the security considerations to apply are the essentially the same. This means that, as well as HTTP, RTSP can be considered an insecure protocol if not used on a underlying secure channel.

In this context, one solution could be represented by the implementation of authentication mechanisms at application layer to be passed in order to access the video stream. However, as many times recalled, IoT devices often show misconfigurations, such as missing password setting or equivalently the usage of default credentials to protect these systems, or the storage of passwords in plain text [59].

In this scenario, attacks targeting IoT devices using this protocol find fertile ground. In particular, RTSP bruteforce attacks target the often inadequate or completely missing authentication mechanisms of devices using RTSP in order to gain unauthorized access to video streams and potentially compromise the security of data generated and handled by the victim. This attack technique can be particularly effective when targeting IoT cameras due to the aforementioned often inadequate security measures in place.

Practically speaking, RTSP bruteforce attacks can be performed by using automatic network enumeration and login cracking tools [63]. In fact, these kind of tools are able to scan the TCP

ports on target systems and, in case a port is detected as opened, they are able to perform automatic exploitation targeting the specific service detected on that port.

Usually, the tool `Nmap`[13] is used for this purpose. It consists of an open source utility for network discovery and security auditing and provides the aforementioned possibility to execute automatic scripts targeting specific services.

In particular, `Nmap` utilizes a script called "drtsp-url-brute"[14] in order to conduct this attack. This script is executed against all the target which are detected to be listening on TCP port 554, typically associated with IP cameras, or any other TCP port where a RTSP service is detected.

The process consists of an attempt to enumerate RTSP media URLs in correspondence of which the multimedia content is provided, through sending several requests to a whole list of common well-known paths on devices such as IoT cameras, thus adopting a bruteforce approach. In particular, `Nmap` utility is equipped with a list of about two hundreds well-known RTSP URLs used by common video surveillance equipment.

Specifically, the script attempts to discover valid RTSP URLs by sending a DESCRIBE request, which is the RTSP method used to ask a server the description of a media object identified by the requested URL, for each URL in the available list. Consequently, it parses the response and, based on this, it determines whether in correspondence of that URL multimedia content is accessible or not. One the location of the multimedia stream was retrieved, because of the aforementioned usual lack of authentication and access control, the eventual attacker could access the multimedia stream.

In conclusion, given the confidentiality that could be required for these multimedia stream and in the light of the above discussed security problems and possible attacks, the need to find appropriate countermeasures arise. One of these can consist of the usage of strong password in order to access the multimedia stream. Additionally, another solution can be represented by network security solution, in order to detect and prevent an ongoing RTSP bruteforce attack. The IDS/IPS module developed in the context of this work, presented in Chapter 6, manages to detect and mitigate this kind of threat, recognizing the traffic generated during the discovery phase and consequently blocking the source of this traffic, thus representing a viable solution.

[13] `https://nmap.org/` [Accessed: Sep 03, 2023]

[14] `https://nmap.org/nsedoc/scripts/rtsp-url-brute.html` [Accessed: Sep 03, 2023]

Chapter 6

Solution Design and Implementation

In this chapter, the designed and developed IoT Proxy, a secure and modular gateway solution for IoT devices networks, will be presented. First, Section 6.1 provides a brief introduction about IoT devices and their security-related aspects, the FISHY platform and the ideas at the basis of IoT Proxy. Secondly, Section 6.2 provides a detailed explanation of how IoT Proxy was designed and implemented in practice, from the architectural point of view and including also relevant developing-related aspects, such as particular data structures or technologies used. Subsequently, the vNSFs supported by the IoT Proxy and the NFV-related aspects are presented in Section 6.2.2. Thereafter, Section 6.3 contains an explanation of the custom IPS vNSF developed as part of the present work, providing an explanation on how the IPS and the oblivious authentication parts are implemented. This Section also includes, in Section 6.3.1, some aspects about network traffic statistics, also providing an overview about the use of this type of data in machine learning. Finally, Section 6.4 explains configuration aspects, indicating the possible modalities to configure IoT Proxy, and providing, in Section 6.4.1, the details about each configuration command provided.

6.1 Overview

One of the most interesting scenarios of the modern Internet is certainly Internet of Things (IoT) [64]. The IoT technologies goes beyond the traditional types of communication (i.e. "human-human" or "human-machine") and enable a different type of communication, which can be identified as "machine-machine", opening the way to interesting scenarios for Internet evolution.

With continuous progress in the development of new technologies, IoT aims to become a ubiquitous global computing network, where everyone, and especially everything, will be connected to Internet.

Overall, each device which is able to convert analogical data into digital information and communicate this data over a network can be classified as IoT. However, it is due to mention that two macro-categories of IoT devices can be identified: the so called Consumer IoT (CIoT) and Industrial IoT (IIoT) [2].

CIoT is more closely concerned with connected objects used by people, such as typical smart home devices like connected thermostats or smart watches. Overall, the objective of CIoT is optimizing the user experience offering a service which can assist the end-user in a certain task.

In contrast, IIoT refers to IoT technologies involved in industrial environments and its main purpose is to create interconnected networks of industrial machines and systems, facilitating the automation of some kind of processes, whit the goal of improving the automation degree and the performances of such systems. It should be noted that, in this context, the term "industrial" can result to be misleading. In fact, the use of these technologies has turned out to be useful for

a wide range of sectors, not necessarily related to the world of industry, such as agriculture or health care.

This brief overview about the IoT world should have given an idea of the potential and the opportunities by this new class of devices. On the one hand, they introduce new features that improve the user experience, supporting the user in daily activities, while on the other hand, they open the way to an increasingly automated and intelligent industry, always intended in broad sense, introducing new possibilities of automation and optimization in the field of work.

In this scenario, the only concerns that stand in the way of IoT massive adoption are represented by the security and privacy aspects [65]. In fact, as aforementioned, IoT devices in general are able to turn certain type of analogical data into digital information and, what is most important in this context, transmit this data over networks, usually Internet. The concerns derive from the fact that, by their constitution, IoT devices are not as powerful as general purpose systems, such as smartphones or laptops. This represents a limit in the implementation of traditional cybersecurity solutions, usually based on computationally heavy and/or memory-intensive operations. Still, these devices are connected to Internet and can potentially handle also sensitive information, such as the voice of a user recorded by a vocal assistant at home, or measures referred to a certain industrial machinery.

In this context, FISHY [1] project, which received funding from the European Union's Horizon 2020 research and innovation program, represents a viable way to address this kind of issues. In particular, the objective of FISHY is the creation of a coordinated cyber-resilient platform with the purpose of establishing trusted supply chains of ICT systems. The platform is based on novel evidence-based security assurance methodologies and metrics, as well as innovative strategies for risk estimation and vulnerabilities forecasting, leveraging state-of-the-art solutions, with the goal of enhancing the cyber-resilience of the whole supply chain.

Referring to the architecture presented in the FISHY project website [2], FISHY platform is not intended to be an incremental integrated solution for cybersecurity, but rather it aims to become an extensible and programmable framework, which can flexibly automate the entire set of ICT systems and security controls handling. The goal is to create an innovative framework, able to provide cyber-resilience to the entire supply chain, in which the performance of complex ICT systems which are involved can be analyzed in terms of security, trust, and privacy impact. To this end, FISHY seamlessly combines advances in several innovative areas, including Network Function Virtualization (NFV), which has been presented in Chapter 2, and machine learning techniques, which have been deepened in Chapter 4.

Modern supply chains consist in many transitions, each one potentially involving many IoT devices. For this reason, FISHY platform provides the presence of a component called "IoT Proxy", which has been projected and implemented in the context of this work. IoT Proxy consists in a modular secure gateway solution for IoT devices networks, and represents the point of contact between the FISHY platform and IoT devices. The idea behind IoT Proxy, and the FISHY project in general, is to outsource security-related aspects management of IoT devices.

In particular, IoT Proxy is concerned with the network security aspects of IoT devices. Since, as aforementioned, IoT devices usually have limited hardware resources which make impossible or really hard to implement traditional cybersecurity measures on them, the idea is to channel all the incoming and outgoing network traffic related to these devices through a potentially much powerful gateway. Moreover, as above cited, IoT Proxy is modular, which means that it is constituted by a set of modules, each one consisting in a specific vNSF that can be switched on and off, creating a cascade of security controls through which the network traffic of the IoT devices is channeled.

Finally, it is due to mention that the proposed IoT Proxy solution can be integrated in the FISHY platform or be used individually as a standalone solution, to provide security functionalities to IoT devices networks, respectively using or not the features offered by other components of the FISHY platform.

[1] https://fishy-project.eu/ [Accessed: May 28, 2023]

[2] https://fishy-project.eu/architecture [Accessed: May 28, 2023]

6.2 Architecture of the IoT proxy

To understand and explain the architecture and the implementation steps which have led to the realization of IoT Proxy, it is first necessary to fully understand the idea behind it. The need for such solution comes from the intrinsic low level of security usually provided by IoT devices, due to the presence of minimal hardware resources on board. It is due to cite that, one possible trivial solution to this, could be to build more equipped and powerful devices which could be able to implement traditional cybersecurity solutions. However, while this approach can be followed when taking CIoT devices into consideration, such as smart televisions or vocal assistants, it turns out to be a non viable solution for IIoT.

In fact, when considering modern industry scenarios, it is not hard to figure out the existence of networks of sensors with hundreds of hosts. If the cost of the single sensor grows significantly, the solution turns out to be no more scalable and no more viable in terms of costs. Therefore, the idea to externalize the security management for this device turns out to be interesting.

As mentioned in Section 6.1, the basic idea behind IoT Proxy is to create a modular secure gateway for IoT devices networks. Hence, IoT Proxy is a gateway, since it is configured as a network device which connects the internal IoT device network to the outside; it is modular, since it offers the possibility to start and stop vNSFs on demand; it is secure, since it provides network security functionalities to IoT devices through the sniffing of all the traffic related to such IoT devices and applying a cascade of vNSFs.

IoT Proxy consists of a coordinator module, which represents the access point to IoT Proxy from IoT devices point of view, and a set of vNSF modules which can be activated or stopped at need, as shown in Figure 6.1. Such activation and stopping can be controlled by a practitioner, intended as a user or automated system, which can send configuration commands to the coordinator, which in turn is in charge of interpreting it and perform the various actions to put in place the requested configuration. IoT Proxy configuration-related aspects are deepened in Section 6.4.

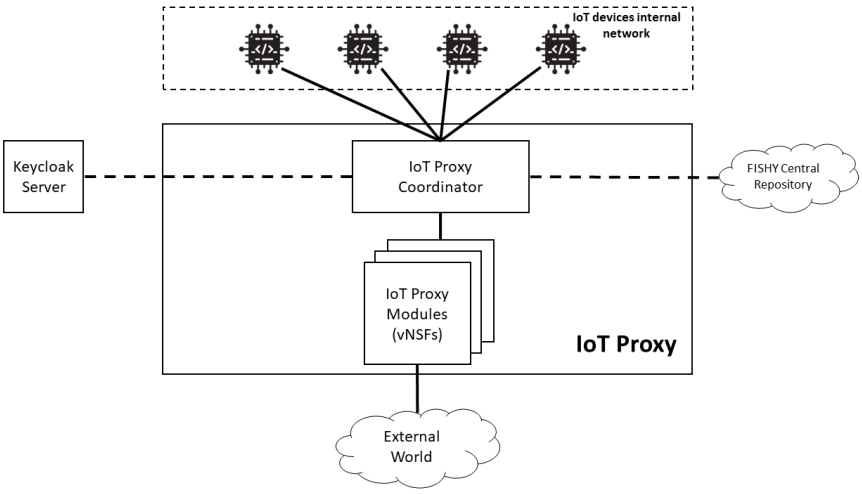

Figure 6.1. Architecture of IoT Proxy

Practically, the proposed IoT Proxy is composed by a coordinator, which consists of a virtualized host which runs Ubuntu[3] 20.04 as operating system, and a set of modules, which are virtualized hosts as well, each one running an associated vNSF. Upon the Ubuntu operating system, the coordinator has iptables and a Python[4] 3.9 interpreter installed, necessary to run

[3] https://ubuntu.com/ [Accessed: Jun 04, 2023]

[4] https://www.python.org/ [Accessed: Jun 04, 2023]

the Python scripts which have been developed for the configuration of IoT Proxy, which will be deepened in Section 6.4.

Therefore, IoT Proxy is a virtualized solution, and specifically it is delivered as a set of Docker containers, to ensure portability of the software and integrability with the FISHY platform, thanks to Docker characteristics. In particular, the coordinator container consists in the aforementioned elements and contains several Python scripts that are necessary for receiving the desired configuration and put it in place, communicating with the host machine which runs the Docker engine. In this respect, a communication channel with the host machine to implement the configurations provided to the coordinator is needed. This is due to the fact that the coordinator process runs in a Docker container and it needs to send commands to other Docker containers, as well as to the host system, which also has to change some networking routes depending on IoT Proxy configuration.

The identified solution for this problem consists in the usage of a named pipe. In Linux-based operating systems, a named pipe, also known as a FIFO, is a special file that works exactly as a traditional pipe but with a name on the filesystem. Multiple processes can access this special file for reading and writing like any ordinary file.

Docker provides the possibility to mount host directories inside containers. Thanks to this, it is possible to share the named pipe between the host system and the container which runs the coordinator. As said, the named pipe works exactly as a traditional pipe, thus implementing also the blocking reading concept. Consequently, the host system listens on the named pipe waiting for something to read, which in this case consists in the shell commands to be executed. The coordinator, instead, when a new configuration is requested, writes the corresponding shell commands in the named pipe.

In this way, the host system loop reads the named pipe and, when a given command to be executed arrives, its process which is waiting on the named pipe reads and executes it. When the configuration to be put in place is related to another Docker container, it is provided by means of `docker exec` commands, which provides the possibility to run an arbitrary command inside a running container from the host system. On the other hand, when the configuration is related to the host itself, the corresponding shell command to be executed is written by the coordinator on the named pipe.

6.2.1 Network interfaces of IoT Proxy

As mentioned in Section 6.2, IoT Proxy is a virtualized solution, delivered by means of a set of Docker containers. The fact of being virtualized provides the possibility to easily arrange an arbitrary number of virtual network interfaces for the containers and to dynamically add and remove them as needed (i.e. on demand). This aspect turns out to be crucial for the implementation of the coordinator, since it needs the presence of several interfaces, both for communication (intended as gateway facilities, such as forwarding traffic) and configuration.

To make the IoT Proxy coordinator receive its configurations, a first network interface, permanently connected to the outside world (which in this context actually means the host system), is needed. It is due to point out that the host operating system could also eventually expose a port to the outside world and bind it to a port of the coordinator container, making the coordinator reachable from the network it is connected to, potentially Internet. Therefore, a first "configuration" network interface is provided for the coordinator. Through this, it can receive configuration commands, directly via API that it exposes or reading them from a message queue, based on how itself is configured. This aspect will be deepened in Section 6.4.

The same configuration network interface is also the one from which the IoT Proxy receives the traffic coming from the IoT devices connected to the network. In this context, all the network traffic from IoT devices received by the machine where the IoT Proxy runs should be forwarded to the coordinator Docker container and consequently to the whole vNSFs chain. This problem is solved by configuring policy-based routing rules.

Policy-based routing is a networking technique that allows to control the flow of network traffic based on predefined policies or criteria, rather than solely relying on traditional routing protocols.

With policy-based routing, packets can be routed according to rules which indicate certain criteria to be met, such as source IP address or destination IP address. Therefore, this technique makes it possible to implement specific routing decisions for different specific conditions.

In the context of IoT Proxy, thanks to it, it is possible to configure specific rules, based on source and destination IP addresses and input interface, on the host machine where IoT Proxy is executed to make all traffic coming from IoT devices to pass through the IoT Proxy coordinator and consequently through all the other active modules, before being sent to the external world through the host machine network card. The same technique is used for the traffic which has a registered IoT device as destination. For this traffic, a specific rule is configured on the host machine and the network packets coming from the external world are forwarded to the last vNSF of the IoT Proxy modules chain, which is fully traced back by the packets to the coordinator, to then actually send the traffic to the IoT device always through the host machine network card. In this way, the IoT devices can reach the external world, and vice-versa, being subjected to the security controls provided.

Besides from the configuration one, another network interface connected to the external world is needed, which takes the name of "outside" interface. This second network interface is needed to forward the traffic related to the various IoT devices connected to IoT Proxy coordinator to the external world and back. Essentially, the outside interface of the coordinator corresponds to the output interface of a traditional network gateway. Through this, the coordinator forwards the IoT devices traffic directly to the external world or to a cascade of vNSFs, as it will be presented in Section 6.4, based on the configuration of IoT Proxy itself.

Therefore, this interface needs to be dynamically configured and this is the reason why the first configuration interface (connected to the external world as well) cannot be used also for this purpose. Actually, from the networking point of view, it could still be possible to handle IoT Proxy configuration and forward IoT devices traffic through a single interface of the coordinator, but in presence of a cascade of active vNSFs, this solution would introduce the passing of additional useless traffic through the vNSF cascade. For this reason, the implementational choice provided this additional outside interface, which is used just to handle IoT devices communications.

Additionally, a third network interface is provided for IoT Proxy coordinator, which is referred as "authentication" interface. This interface is provided for the communication of the coordinator with the Keycloak[5] server, which is used by IoT Proxy for the authentication of API calls it receives. Through this interface, the coordinator can communicate with the Keycloak server, which is responsible for the API call authentication. In this way, if the practitioner performing API calls provides a valid access token, it will be allowed and the execution of the corresponding function of the coordinator will be triggered. Otherwise, the Keycloak server detects an error and blocks the execution of the coordinator function. Keycloak principles and functioning will be presented in Section 6.4.2.

Summarizing, IoT Proxy coordinator counts the presence of several network interfaces, each one dedicated to a specific purpose. Among them, there are two static network interface, used to receive configurations and to communicate with the API authentication server. Moreover, a third dynamically configured network interface is provided, which plays the role of output interface for the traffic related to IoT devices. Finally, it should be noted that the configuration interface is also used to forward traffic coming from connected IoT devices to the IoT Proxy coordinator Docker container.

6.2.2 Supported vNSFs

As introduced since the beginning of the present Chapter, IoT Proxy is a modular solution. Practically, this means that the security controls offered by IoT Proxy are implemented by means of a set of modules, each one implementing a specific vNSF and each one that can be switched on and off at need, building a cascade of vNSFs which are queued down the IoT Proxy coordinator.

[5] https://www.keycloak.org/ [Accessed: Jun 11, 2023]

The final version of IoT Proxy delivered in the context of the present work, supports two different vNSFs: IPS and strongSwan. IPS module deals with intrusion prevention and its work consists of recognizing in a proactive way malicious activities inside a network by observing the network traffic in transit on it, with the purpose to block them to mitigate potential threats. In particular, the proposed IPS module is able to recognize two different types of attacks: flooding and RTSP[62] bruteforce. These two attacks are relevant in this context, since, as presented in Chapter 5, when targeting IoT devices they turn out to be particularly effective.

The proposed IPS module implements also an innovative authentication mechanism, which is based on the usage of machine learning techniques and aims to detect anomalous scenarios through the analysis of network traffic statistics, while strongSwan consists of an IPsec-based VPN terminator, which allows the creation of IPsec VPN tunnels. Since it represents a crucial part in the context of the present work and it has been developed right in this context, IPS module will be deepened to a sidebar, in Section 6.3.

The modular nature of IoT Proxy allows it to be easily extensible. In fact, at the state of the art, the proposed solution supports the two aforementioned vNSFs, however, any vNSF which turns out to be implementable in the form of Docker container can be integrated in IoT Proxy and is potentially supported by it.

Since IoT Proxy generally manages the vNSFs cascade independently from the nature of the involved vNSFs, in order to extend the vNSFs supported by IoT Proxy, it is just necessary to append a few lines of Python code. In fact, IoT Proxy functioning is governed by information stored in the coordinator internal data structures. One data structure of these consists of a Python dictionary which maps vNSF name, used as parameter in configuration commands, to the correspondent Docker image to be used when the related container must be executed.

Therefore, to integrate the support for an additional vNSF, it is just necessary to update this particular data structure in the code, appending the desired key-value pair. In this way, the mapping between the new vNSF name, which will be used as parameter in configuration commands to refer to the new vNSF, and the related Docker image, which will be used to execute the corresponding container, is made known to IoT Proxy coordinator. From that moment on, IoT Proxy completely supports the newly added vNSF and will be able to perform the usual start/stop operations on it.

In this way, the newly added vNSF is fully supported and treated like a generic one, from IoT Proxy point of view. Eventually, dynamic configuration may be needed for the new vNSF, which means something that cannot be embedded in the (static) Docker image but rather depends on IoT Proxy environment state, such as the content of the IPS module configuration file, which depends on the registered IoT devices. If so, it is possible to integrate the Python code which performs the necessary actions in IoT Proxy coordinator vNSF activation and device registration functions, as well as code for eventual necessary cleanup operations can be added within the coordinator vNSF stopping and device removal functions, as it was done for IPS module. Finally, it is due to remark that each command which is not related to the coordinator itself must be delivered to the recipient vNSF or host system through the named pipe.

strongSwan

strongSwan is one of the supported vNSF by the IoT Proxy version delivered in the context of this work. It consists of a complete IPsec VPN solution and provides encryption and authentication to servers and clients. As presented in Chapter 5, it offers features for establishing a secure communication channel between two entities, which means devices or whole networks, thus introducing the possibility to communicate over the public Internet as it is done in private networks, implementing the idea at the basis of a VPN.

strongSwan is based on the IPsec protocol suite, which operates at the IP layer to enhance network communication security by providing confidentiality, integrity, and authentication to IP packets. This solution supports all the VPN types, providing robust encryption, authentication, and key exchange mechanisms. It empowers VPN connections through modern encryption algorithms, together with strong authentication methods like PKI (Public Key Infrastructure) and

pre-shared keys. The IKEv2 protocol is leveraged for aspects related to cryptographic algorithms parameter negotiation, session key generation, and actor authentication.

Additionally, strongSwan can be deployed using Docker containers, leveraging the benefits of lightweight virtualization, presented in Chapter 3. The process involves creating a Docker image containing the necessary dependencies, IPsec components, and strongSwan configurations. The setup involves generating certification authority keys and certificates for gateway authentication, followed by creating similar credentials for the gateway itself.

In the context of this work, strongSwan image, which can be build from the provided Dockerfile, comes complete with certificates and keys for certification authority and gateway itself. Moreover, a configuration file `/etc/ipsec.conf` and a secrets file `/etc/ipsec.secrets` are provided, which are necessary, respectively, to configure the parameters and to define the users of the VPN. Subsequently, when executed in the context of IoT Proxy environment, strongSwan vNSF can be configured via IoT Proxy configuration commands for strongSwan, presented in 6.4.1, with the purpose to create a VPN of IoT devices.

6.3 IPS vNSF

The proposed IPS module, as introduced in Section 6.2.2, aims to recognize in a preventive way the potential malicious activities inside an IoT devices network through the observation of the network traffic in transit on it. Therefore, as explained in Chapter 5, the IPS module purpose is to prevent the damages that malicious activities that target IoT devices can cause, by anticipating them. This proactive behavior is possible thanks to machine learning techniques, which allow to recognize malicious activities in advance by analyzing the network traffic, with the ability of identifying known traffic patterns that can match the ones typical of an attack.

In IoT environments, real-time threat detection turns out to be particularly important to promptly respond to security incidents and prevent further damage to the devices themselves and to the network. The design philosophy of IoT devices focuses on minimizing power consumption and cost, resulting in constrained hardware resources. These limitations pose challenges when deploying traditional security measures, as they often require considerable computational power, which may exceed the capacity of IoT devices. Consequently, the presence of an IPS turns out to be crucial for protecting the IoT devices from the potential threats.

In the context of the present work, the proposed IPS is able to recognize the network traffic pattern of two different types of attacks, namely flooding and RTSP bruteforce, and consequently actively block the attacks. As discussed in Chapter 5, flooding consists in carrying out a DoS attack, which in the context of the present work was performed through the use of `LOIC`[6]: a software that was originally developed to test the response of systems to large quantities of requests, nowadays used also by attackers to perform DoS attacks. RTSP bruteforce, on the other hand, is performed by using network enumeration and login cracking tools, usually `Nmap`, with the purpose of finding the RTSP URLs of IoT cameras by brute-forcing and consequently attack the discovered endpoints to attempt to access live streaming content.

The aforementioned kinds of attacks turn out to be particularly effective on IoT devices. With respect to flooding, which is an attack of DoS type, when targeting IoT devices, it appears to be particularly harmful because of the typical lack of resources in IoT devices, that can be therefore saturated relatively easily. On the other hand, when considering RTSP bruteforce, it adapts perfectly to IoT scenarios since the ultimate purpose of the attacker performing this malicious activity is to access live video streaming data and the frequent lack of adequate security measures in IoT devices, including cameras, simplifies the work of the attacker.

In general, an IPS provides an additional layer of security to network environments with respect to simple anomaly detection, taking a step further by actively recognizing, blocking and mitigating malicious activities in advance before they can cause significant damages. This proactive approach

[6] https://sourceforge.net/projects/loic/ [Accessed: Aug 16, 2023]

turns out to be particularly important in the context of the attacks treated in the present work, especially when considering flooding. When considering DoS attacks targeting IoT devices, the timely blocking of the attack is fundamental because, always because of the scarcity of resources, it may take a short time to stop the service or cause irreversible damage to the IoT device itself performing this kind of attacks.

The proposed IPS module also implements the concept of "oblivious authentication". In cybersecurity, authentication consists in the process of verifying the identity of a user, a process, a device, or in general a communication actor. Traditionally, authentication is accomplished via the demonstration of some kind of proof, which ensures that the party involved actually is who or what it declares to be. This proof can consist in the knowledge of a secret, like a traditional password, but also on more complex and innovative alternatives such as a biometric reading of a fingerprint.

Oblivious authentication, on the other hand, follows a different approach which aims to authenticate entities, represented by IoT devices in the context of the present work, through the analysis of statistics calculated over the network traffic related to such devices, exploiting machine learning techniques. Through the extrapolation and analysis of network traffic statistics, it could be possible to identify whether the characteristics of network traffic related to a given IoT device are in line with the typical traffic profile expected for it or if they deviates from it.

In this way, assuming the ideal case in which the underlying machine learning engine provides perfect accuracy, a device could be totally identified by its specific network traffic profile and alerts/errors can eventually be generated in presence of anomalous situations or events, with the purpose of promptly take action and mitigate the eventual threat.

This kind of innovative authentication method turns out to be particularly suitable for IoT scenarios, allowing to completely overcome the limitations introduced by the lack of hardware resources of IoT devices. In fact, in this way the computing and memory load of authentication process are entirely externalized to the machine which runs the machine learning model, while all IoT devices do is to perform the task they are designed for, collecting, processing and sending data through the network.

To fully take advantage of IPS and oblivious authentication features, from an architectural point of view, the most convenient choice is to implement it in a network gateway, such as it is done with IoT Proxy, forcing all traffic related to involved IoT devices to pass through it and consequently be analyzed. In this way, in principle, the identity of each IoT device could be entirely retrieved from its related network traffic, thus removing the need for, often insecure, cryptographic protocols for IoT devices.

In the context of the present work, oblivious authentication idea was implemented as part of the custom IPS module integrated in IoT Proxy. The implementation of an oblivious authentication mechanism consists in several steps which, starting from a knowledge base which consists of a set of network traffic capture files, lead to the creation of the IPS vNSF. In fact, the first point which has to be taken into consideration when designing this vNSF is the choice of the data source to use, which essentially means the starting dataset to feed the underlying machine learning model. With respect to this, the CIC IoT Dataset 2022[7] turns out to be particularly suitable.

The aforementioned dataset, which has been started and is maintained by CIC[8], collects several capture files which were realized capturing the traffic related to forty IoT devices of different type and category, as well as to ongoing attacks targeting those devices. The available capture files provides network traffic details related to different contexts in which the capture happened. In particular, captures were performed while individually powering on all the IoT devices, while they were in an idle state, while individual functionalities of devices were activated, while devices were active and interacting with a user and while devices are targeted by one of the presented attacks.

With respect to the oblivious authentication part, first, in the context of the present work, a preliminary operation is performed on the available dataset, since network traffic related to

[7] https://www.unb.ca/cic/datasets/iotdataset-2022.html [Accessed: May 30, 2023]

[8] Canadian Institute for Cybersecurity

idle and active scenarios were provided divided by date and not for single device. To implement the oblivious authentication idea, the dataset must be labeled using device category or device type as labels, so that the machine learning model can be trained observing correct associations. Therefore, from each capture file belonging to idle and active scenarios, one capture file for each device is extracted. Since a specific and fixed MAC address is assigned to each IoT device during the data collection, this extraction is possible filtering packets by specific MAC address, leveraging functionalities offered by `tshark`[9] command line tool.

After this preliminary operations, the data base, intended as the necessary initial data, is definitely prepared. At this point, it is due to remark that the IPS vNSF does not work with raw captures of network traffic, but rather it is based on traffic statistics, each one representing a starting feature for the underlying machine learning model. Therefore, it is necessary to extract network traffic statistics from the capture files.

For this purpose, in the context of the present work, `Tstat`[10] tool is used. `Tstat` is a tool developed by Politecnico di Torino and it consists of a passive network traffic sniffer, which is able to provide insights on the traffic patterns at both the network and the transport levels, providing statistics related to the network traffic it analyzes. Providing a capture file or a network interface to sniff on as input, `Tstat` is able to extrapolate more than forty network traffic statistics related to packets it observes. The concept of network traffic statistics and the set of statistics calculated by `Tstat` will be presented in Section 6.3.1.

At this point, `Tstat` is executed against each single capture file, extrapolating aggregated statistics from it and, contextually, the actual dataset which is used to train the machine learning model is populated. For each single file provided as input to `Tstat`, the tool outputs several data frame of statistics, to which a label is added. The label represents the category or the device type associated to that capture file, for the oblivious authentication part, or the type of attack to which that traffic is associated, with respect to the IPS tasks.

With respect to oblivious authentication concept, in the present work, two different machine learning models are trained and then integrated in the IPS vNSF, one which aims to distinguish device categories and another which has the purpose of recognizing device types. It is worth remarking that, from this point on, the steps to be performed for the training of the two machine learning models are absolutely the same.

With respect to the IPS features offered by the present module, the capture files related to flooding and RTSP bruteforce attacks present in the CIC IoT Dataset 2022 are analyzed individually using `Tstat`, as well as all the files related to non-malicious traffic from IoT device, and the consequent resulting traffic statistics records are labeled as “Flood” or “RTSP”, respectively. Therefore, in this phase, for both the category and device type recognition models, two additional classes are provided, to enable the machine learning model to classify attack-related traffic.

Considering the model for the recognition of categories as reference, during the completion of this process, which is repeated for each capture file available, all `Tstat` output records and associated category are saved in a file, so that, after the process is completed, all the data necessary to train the machine learning model are ready and well-formatted, which means the construction of the dataset to be used for the training of the machine learning model is completed.

It is due to mention that not all the traffic statistics provided by `Tstat` are used to train the machine learning model. In particular, intuitively irrelevant information, like whether the client had an internal or external IP address, are not considered. Moreover, some kind of information, namely IP addresses and ports used and the absolute time associated with the first and last packet in TCP flows, represent a clear risk of overfitting, since the value of these attributes is strongly tied to the specific training data and could negatively influence the generalization property of the model. For this reason, these data were cut off the dataset as well, before it is processed by the machine learning model.

[9] https://www.wireshark.org/docs/man-pages/tshark.html [Accessed: Jun 12, 2023]

[10] http://tstat.polito.it/ [Accessed: Jun 12, 2023]

Moreover, a feature pruning process is performed on the available data, identifying and removing the features which turn out to not provide significant information to the model. Essentially, plotting for each feature the distribution of its values, it is possible to identify cases in which the set of different values assumed by the specific feature is significantly small or even consists of a single value. In this case, it is reasonable not to use such a feature, as it does not provide a useful element to the model to learn to distinguish elements of the various classes.

In the context of the present work, the choice of the specific machine learning model to be used is quite straightforward, and the choice falls on random forest algorithm. As already presented in Chapter 4, random forest has demonstrated great performance in solving similar problems and, as a general characteristic, it shows intrinsic capability of dealing with multi-class classification. In practice, the implementation of the random forest algorithm used is the one provided by scikit-learn[11], an open-source Python library which offers machine learning functionalities.

Before the training phase is started, the training and test sets are built splitting the original dataset. At this point, a random forest model is trained feeding it with the available training set. In this case, no particular configuration is provided to the random forest, which is generally configured with the default settings from `scikit-learn`. The only two customization in the configuration consist in the usage of ten estimators compared to the one hundred which are set by default, for performance reasons, and the usage of the `class_weight=‘‘balanced"` attribute, since the classes in the dataset turn out to be quite unbalanced.

After the training phase is completed, a test phase is performed on the previously generated test set. In this way, it is possible to extract a set of metrics, calculated during the test phase, with the objective to evaluate the performances of the trained multi-class classifier. Experimental results of this phase, as well as the results obtained in the previously introduced features pruning process, are presented in Chapter 7. Finally, the trained model is saved to a file, by using the dump feature offered by `Joblib`[12] Python module. In this way, the trained machine learning model can then be loaded into the Docker container executing IPS vNSF. As said, the same process is repeated for the model used for the recognition of device types.

In conclusion, the saved trained models are included in the IPS Docker image and, when the associated container is executed, they are loaded by the sniffing script which is executed in the context of the Docker container running IPS vNSF. The sniffing script is responsible for the actual controls on network traffic of IoT devices connected to IoT Proxy. Practically, it sniffs the network traffic, extrapolates live statistics related to it using `Tstat` and feeds the trained random forest with those statistics, asking for a prediction.

Consequently, the predicted category or device type is compared to the expected one for the specific device involved in that traffic. In case the prediction matches the expected value, no actions are performed and the IoT device is considered as correctly authenticated. Instead, if the prediction made by the random forest and the expected value for the device differ, two different scenarios are possible.

In fact, it is worth remarking that, because of the nature of the machine learning models at the basis of the proposed IPS module, two different types of anomalous behaviors can be recognized. One of the possible scenarios is the one in which the model recognizes a network traffic profile referable to the one typical of a known attack. In this case, the model classifies the traffic as malicious and the source of this traffic is isolated from the network through the configuration of an appropriate `iptables` rule, blocking all the related traffic.

On the other hand, a device can behave unexpectedly, presenting a traffic profile not consistent with the expected one but still not classified as an attack, therefore being categorized by the model as a different category or device type. In this case, an alert is raised to signal the anomalous behavior or the traffic related to that device is completely dropped from that moment on, based on how the sniffing script was configured.

[11] https://scikit-learn.org/ [Accessed: Jun 13, 2023]

[12] https://joblib.readthedocs.io/ [Accessed: Jun 13, 2023]

When the IPS vNSF is stopped, consequently to the receiving of the correspondent configuration command by IoT Proxy coordinator, eventual firewall rules for traffic dropping are automatically removed.

6.3.1 Traffic statistics

In the context of the networking world, to extrapolate network traffic statistics can turn out to be a crucial activity. By way of example, the medium number of incoming TCP [60] connection for a given server or the total amount of data packets sent on TCP connections from a given subnet could turn out to be important information that could help in network dimensioning decisions. This kind of aggregated data could be useful when performing analysis during network engineering process, since they represent important metrics that allow to properly set some parameters peculiar to the network dimensioning.

Especially with the advent of modern Internet, and the contextual appearance and widespread of technologies as IoT, data intensive application and cloud-based applications, the demand for network facilities, especially with performance guarantees, has been massively soaring [66]. This new needs coming from modern technologies increases the importance of precise and representative network traffic statistics. For this reasons, understanding how to carry out network traffic statistics efficiently and accurately has become one of the focuses of research.

Besides from helping in the network engineering process providing useful aggregated metrics, traffic statistics turn out to be also an interesting source of data for machine learning and, in general, artificial intelligence. Nowadays, for example, it is quite common to find a so called Intrusion Prevention System (IPS)[13] in an organization network, which aims to prevent malicious activities inside a network and anticipate them.

Precisely because this kind of system has to prevent the malicious activity, it must own some kind of intelligence, in order to understand a risky situation in advance, based on some kind of distinctive feature of the eventually malicious network traffic. Therefore, this kind of systems' core is based on machine learning and artificial intelligence mechanisms and learn to recognize malicious activities inside a network through the observation of past malicious network traffic. This is a typical example of how the network traffic statistics can be used to define metrics which are then used to characterize a certain type of, in this case malicious, traffic.

Likely, using network traffic statistics rather than individual network packets in this process, allows machine learning models to perform a learning process based on the observation of aggregated, instead of raw, data. Accordingly, the machine learning model will develop a knowledge of the bigger picture, which likely improves the ability to generalize. On the contrary, empirically reasoning, using raw network traffic to train machine learning models could lead to a too punctual learning by the model.

In this context, a useful tool to extract aggregated statistics from network traffic is represented by Tstat[14]. Started as evolution of TCPtrace[15], Tstat is a tool able to produce statistical data from network packet traces, obtained either from live capture on an interface or from a traffic capture file. The tool is able to extract a comprehensive set of statistics. For example, one of the offered feature consists in streaming classification, and is able to identify the type of video format which is used during a streaming process.

In the context of IoT Proxy, and in particular of IPS vNSF, data of interest consist in a subset of the core TCP statistics extrapolated by Tstat. As explained in Section 6.3, not all the statistics generated by Tstat are used in the context of the present work, since they turn out to be clearly irrelevant or introduce a clear risk of overfitting.

[13] An IPS is a network security tool that continuously monitors a network and take several actions to prevent malicious activity inside it.

[14] http://tstat.polito.it/ [Accessed: Jun 15, 2023]

[15] http://www.tcptrace.org/ [Accessed: Jun 15, 2023]

The set of Tstat statistics which are used in the context of training the machine learning model for IPS vNSF, are indicated in Table 6.1. The used statistics are are the result of the above mentioned reasoning and of a process of feature selection, which has been introduced in Section 6.3 and will be deepened in Chapter 7.

Name	Description
packets	total number of packets observed form the client/server
RST sent	0 = no RST segment has been sent by the client/server
ACK sent	number of segments with the ACK field set to 1
PURE ACK sent	number of segments with ACK field set to 1 and no data
unique bytes	number of bytes sent in the payload
data pkts	number of segments with payload
data bytes	number of bytes transmitted in the payload, including retransmissions
rexmit pkts	number of retransmitted segments
rexmit bytes	number of retransmitted bytes
out seq pkts	number of segments observed out of sequence
SYN count	number of SYN segments observed (including rtx)
FIN count	number of FIN segments observed (including rtx)
C first payload	Client first segment with payload since the first flow segment (in ms)
S first payload	Server first segment with payload since the first flow segment (in ms)
C last payload	Client last segment with payload since the first flow segment (in ms)
S last payload	Server last segment with payload since the first flow segment (in ms)
C first ack	Client first ACK segment (without SYN) since the first flow segment (in ms)
S first ack	Server first ACK segment (without SYN) since the first flow segment (in ms)

Table 6.1. Tstat traffic statistics used in IoT Proxy

6.4 IoT Proxy Configuration

After having introduced general aspects and architecture of IoT Proxy and supported vNSFs, it is due to present IoT Proxy different setups characteristics and how the managed vNSFs and IoT devices can be configured. IoT Proxy receives its configuration commands from the network. In this context, receiving a configuration command means to trigger a change in the state of the controlled vNSFs and/or the IoT Proxy coordinator itself. An example of configuration, which will be deepened in the present Section, is represented by the registration of a new IoT device to IoT Proxy.

In general, when IoT Proxy coordinator receives configurations, two macro-actions that are carried out can be identified. The first action concerns Docker-related aspects, which means all the involved Docker containers and networks are created/deleted. Secondly, all the necessary IP [67] routes and routing rules are updated on the involved hosts, to maintain functioning and connectivity.

In the final version of the implementation delivered in the context of this work, when executed, IoT Proxy coordinator shows three network interfaces, obviously connected to five different networks. This initial setup is realized by means of docker-compose[16], which allows to arrange a

[16] https://docs.docker.com/compose/ [Accessed: Jun 06, 2023]

virtual environment constituted by Docker containers.

Docker-compose is a tool used to simplify the management of multi-container applications in Docker environments. It allows to define and configure multiple Docker containers, their settings, and how they interact each other, for example from the networking point of view, in a single configuration file. This helps developers orchestrate complex applications with various interconnected services, making it easier to deploy application composed by multiple Docker containers.

The three network interfaces consist of configuration, outside and authentication interface, each one which having its specific purpose, as presented in Section 6.2.1. The initial setup is represented in Figure 6.2.

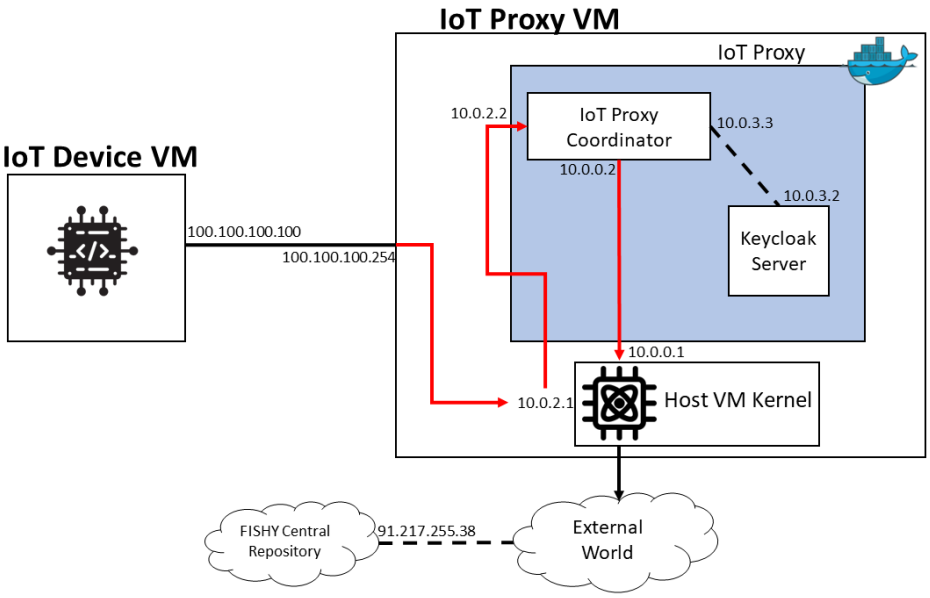

Figure 6.2. Initial IoT Proxy environment configuration

As introduced in Section 6.1, IoT Proxy can be used as a component of the FISHY platform or as a standalone solution. In practice, this difference is reflected in the fact that IoT Proxy offers two alternative mechanisms for receiving configurations. It is due to mention that there are no changes in the way IoT Proxy works if the former or the latter mechanism is used, unless the channel used by the practitioner to send configurations and used by IoT Proxy coordinator to receive them.

When used as a standalone solution, at application level, IoT Proxy coordinator runs a Python Flask[17] application. Flask is an open source lightweight framework for Python which allows to create web applications. IoT Proxy does not provide a front end interface, therefore, when in this configuration, it consists of an HTTP [68] server listening on port 5000 of the configuration interface and exposing a set of APIs.

To send API requests to IoT Proxy, these requests must be authenticated, and here is the point in which Keycloak comes into play. When IoT Proxy coordinator is contacted by a client, an authentication token must be provided along with the request, as a query parameter. Keycloak server verifies the submitted token and determines whether the request comes from an authenticated user or not. If the token is correct, then the actual code provided by the coordinator is executed, otherwise an error message is returned to the client. Keycloak-related aspects will be deepened in Section 6.4.2.

[17] https://flask.palletsprojects.com/en/2.3.x/ [Accessed: Jun 06, 2023]

In general, when APIs are used, the configuration command and the related parameters are part of the URL contacted with the purpose to execute that specific command. Therefore, all API calls will consist in a GET HTTP request to a URL in the following form: `http://<IoT_Proxy_address>:5000/<API_command>/<parameter_1>/<parameter_2>/.../?access_token=<authentication_token>`, where the URL parameter `access_token` is always passed to authenticate the request.

On the other hand, when used as a component of the FISHY platform, IoT Proxy coordinator reads configurations from the FISHY central repository. Central repository is a repository on the web which stores configuration commands for all FISHY platform components, which can access them through a message queue[18]. Therefore, when in this configuration, the coordinator actually consists of a Python application that continuously listens on the message queue waiting for configuration commands.

The interfacing with the message queue is done through RabbitMQ[19], a message broker which implements the AMQP protocol[20]. When a practitioner pushes a configuration on the central repository, thus acting as the publisher in the jargon of message queues, this configuration is stored in the central repository. Consequently, IoT Proxy coordinator module, playing the role of the consumer of the queue, receives this message. Configuration commands must be published to the message queue, by means of a POST HTTP request to a specific path of the central repository endpoint. The URL to be requested for this purpose is `https://fishy.xlab.si/tar/api/reports`.

To push a configuration command, the POST request body must contain a record in json[21] format, which is composed by two fields: source and data. Source field could be used to specify a string indicating who pushed the configuration command, while data value must be an XML [69] object composed by a single XML tag, which must contain a string representing the command to be executed.

Among its attributes, the XML tag must contain `nsf`, which must be set to `iot_proxy` to send commands to IoT Proxy. When IoT Proxy coordinator, which is waiting on the message queue, sees a new message, it checks if the nsf attribute is set to that specific value and, thanks to this, it is able to understand that that command is for it. Consequently, it parses the command and executes the corresponding actions.

Sending a configuration command through the central repository actually means to send a POST HTTP request to the following URL: `https://fishy.xlab.si/tar/api/reports`. The POST request must include the header `Content-type`, with its value set to `application/json`. The body of the request, as introduced above, must contain a json-formatted string with the following structure:

```
{
  "source" : "<source>"
  "data" : "<XML_LLCONF_TAG>"
}
```

where `XML_LLCONF_TAG` is structured as follows:

```
<llconf nsf="iot_proxy" format="ASCII-text" filename="iot_proxy.config">
  CONFIGURATION_COMMAND
</llconf>
```

[18] A message queue is a form of asynchronous service-to-service communication. Messages are stored in the queue until they are elaborated and deleted.

[19] `https://www.rabbitmq.com/` [Accessed: Jun 06, 2023]

[20] `https://www.amqp.org/` [Accessed: Jun 13, 2023]

[21] `https://www.json.org/json-en.html` [Accessed: Jun 07, 2023]

It is due to cite that, when using IoT Proxy as a standalone solution, it is possible to provide an initial configuration file. This configuration file is read by the coordinator before starting the web server and can contain a series of commands, one per line, in the same form of `CONFIGURATION_COMMAND` used with central repository.

Finally, it is worth remarking that the internal operations to put in place the configuration changes required by commands does not change at all in case the IoT Proxy is used as a standalone solution or not. In fact, the only aspect that changes with these two alternatives is the way in which configuration commands are received and the way in which the related parameters are parsed. After this point, the internal functioning is exactly the same, and it is managed thanks to some IoT Proxy coordinator internal data structures (i.e. Python lists and dictionaries) which store the state of the whole environment.

6.4.1 Configuration commmands

After having provided general information on IoT Proxy configuration, it is worth explaining the various configuration commands that are used to configure IoT Proxy coordinator and the surrounding vNSFs and IoT devices. Four generic configuration commands are provided to register/remove an IoT device and activate/stop a vNSF. In addition, two configuration commands specific for strongSwan vNSF are available. For each one of the below configuration commands presented, the API path and the correspondent command to be included in the XML tag, when using the central repository, or in the IoT Proxy configuration file when using APIs, are provided.

IoT device registration

- API: `/register_device/<Device_IP>/<Category>/<Device_Type>?access_token=<authentication_token>`
- central repository: `ADD_IOT_DEV <device_IP> <device_category> <device_type>`

This configuration command allows to register a new IoT device to IoT Proxy. The API path and the central repository command are actually built using three parameters, which consist of the device IP address, the device category and the device type. The device IP address trivially consists of the IP address attributed to an IoT device which is connected to the internal IoT network. The category and the device type are used to associate the registered device with its category and type, which will be used by the IPS vNSF, when active, to know the network traffic profile typical of the device, with the purpose of implementing the oblivious authentication part.

First, when this command is received, IoT Proxy coordinator opens the named pipe shared with the host system in write mode, so that all the necessary commands to be executed by the host system to put in place the requested configuration can be sent through the named pipe. Subsequently, it performs a series of controls, checking if all the necessary parameters have been provided, if the device type indicated actually belongs to the specified category and if the device IP address has been not yet assigned to another device.

It should be noted that the functioning of this configuration command is tied to the assumption that the IoT device had the machine where IoT Proxy is executed configured as default gateway for it network communication. Subsequently, an IP route is added to all the eventually active vNSFs as well as to the host system, to make them know how to reach the new device.

Moreover, when this configuration command is triggered, a set of routing rules are configured on the machine that hosts the IoT Proxy. In particular, a first rule is configured to indicate that all the traffic coming from the IoT device must follow the routes indicated in the “iotproxy” routing table, which is created for the purpose, and a route which sends by default all the traffic to the IoT Proxy coordinator is configured in this new routing table. Similarly, a second rule is specified to make all traffic coming from the external world and intended to the IoT device to follow the routes contained in the “ext-to-iot” routing table. Last, a route which specifies the last vNSF in the chain as the default gateway is configured in this table, so that all traffic coming

from outside and intended to the IoT device traces back the IoT Proxy chain before reaching the destination.

Finally, if the IPS vNSF is active at the moment of the registration of the new IoT device, a line is appended at the end of the configuration file of the vNSF itself. This insertion is necessary to notify the IPS vNSF of a new IoT device registration, with the purpose of letting it know the device IP address, category and type, so that the traffic analysis must be performed for the newly added device.

IoT device removal

- API: `/remove_device/<iot_device_name>/?access_token=<authentication_token>`
- central repository: `REM_IOT_DEV <device_name>`

This configuration command allows to remove a previously registered IoT device from IoT Proxy. In this case, the API path and the central repository command are built using just one parameter, which consists in the device IP address that was indicated, during the registration, for the IoT device to remove.

Therefore, the execution of this configuration command is quite straightforward and actually provides only three actions. These three actions are, in chronological order, the opening of the named pipe shared with the host system, the insertion of a firewall rule in IoT Proxy coordinator Docker container to drop all traffic related to this device and the removal of the routes to the IoT device for all the involved hosts.

vNSF activation

- API: `/activate_vnsf/<vNSF_Name>?access_token=<authentication_token>`
- central repository: `START_VNSF <vNSF_name>`

As aforementioned in Section 6.1, IoT Proxy is a modular component, which is able to support a cascade of multiple vNSFs. The present configuration is the one which allows to start the execution of a new vNSF, which means the insertion of an additional level to the security controls for the IoT devices traffic. The API path and the central repository command are built using just one parameter, that is the name of the vNSF to be activated.

IoT Proxy coordinator internally holds a mapping between the vNSF name and the correspondent Docker image to be used when executing the container which implements that specific vNSF. The final version of the implementation of IoT Proxy made in the context of this work supports two different vNSF, which in this context are referred with the names of “IPS” and “strongswan”. In this context, each vNSF shows at least two interface, namely “top” and “bottom” interface.

As usual, when executing this command, the first actions performed are the opening of the named pipe shared with the host in write mode and checks, in this case verifying that the requested vNSF is not already active. Secondly, a new Docker network, which will exclusively host the newly started vNSF and the preceding one, or IoT Proxy coordinator, is created. This restriction is the basis which makes the implementation of the vNSFs cascade realizable and deserves particular attention.

When activating vNSFs, the last vNSF of the cascade plays the role of the gateway for the whole IoT Proxy environment. Therefore, if more than one vNSFs are connected to the same network, configuring IP routes in the various hosts to force traffic pass through a specific device, since the whole cascade must be traversed, would result in redirection errors[22]. For this reason

[22] a redirection error is generated when a host notice that a received network packet has to be forwarded to the same network interface it was received from. Consequently, an ICMP [70] Redirect is returned to the sender, which indicates the sender to directly communicate with the destination host in future.

the approach followed is to build virtual networks containing only two hosts, which in this case are two vNSF or a vNSF and the coordinator. For simplicity, in this context, the coordinator is considered a vNSF as well.

In a similar way of what happens for IoT devices, the subnet associated to the new Docker network is assigned in an incremental way, which means the first vNSF network created corresponds to 20.0.4.0/24, and for subsequent networks the third octet always increments at any new vNSF activation. Imaging the cascade of vNSF from top to bottom, the top vNSF always has the value 2 in the last octet of its IP address in the new network, while the bottom vNSF always gets value 3.

Consequently, a Docker container is executed using the Docker image associated to the requested vNSF, assigning to the container the name specified in the configuration command. For the final version delivered in the context of this work, two names are admitted: `IPS`, which is associated to the IPS vNSF, and `strongswan`, which is obviously associated to the strongSwan vNSF. Otherwise, the IoT Proxy returns a error to the requester, specifying that it does not handle the requested vNSF.

The container is then connected to the newly created network as "bottom" vNSF. Depending on the type of vNSF to be activated, different actions are performed. If IPS, a line for each one of the already registered IoT devices is written inside the configuration file of the vNSF, to notify it of the presence of these IoT devices. On the other hand, if strongSwan activation is requested, preliminary actions to setup the VPN terminator, such as keys and certificates creation, are performed. In addition, when strongSwan is activated, the container which runs it is also connected to an additional network, thus providing an additional network interface, used as IPsec VPN terminator, and further commands are executed on the host to expose the ports related to a VPN communication from this container interface to the external world, to allow connections.

Subsequently, the bottom interface of the last vNSF of the cascade is disconnected from the external network and is attached to the newly created network as "top" vNSF, while the bottom interface of the newly started vNSF is attached to the external network, taking the place of the previous one.

Later, the IP routes of the host system are replaced to make it pass via the newly started vNSF to contact IoT devices, the default gateway of the "top" vNSF is updated and set to the address of the newly started vNSF on the network they both reside, and IP routes toward IoT devices are added to the started vNSF.

Finally, NAT [71] functionalities are disabled on the "top" vNSF and enabled on the new last vNSF of the cascade, via iptables commands. This is due to the fact that, when forwarding traffic outside the IoT Proxy environment, NAT is compulsorily needed, while, for the "internal" vNSF in the cascade, NAT is not only unnecessary but must be necessarily disabled. This is because the vNSFs in the cascade may need to associate determinate network traffic to a specific device, as in the case of IPS. If the previous vNSF masqueraded the device IP address, the next vNSF would not be able anymore to associate given traffic streams to a specific IoT device.

It is worth remarking that the activation of a cascade of vNSFs represents a critical step in case IoT Proxy is configured through the FISHY central repository. In fact, when in this mode, IoT Proxy coordinator is continuously listening on a message queue available on the web and, while the connection remains the same for what the application level perceives, at lower level there is a change in the network configuration, which implies to start a new connection to the message queue. This may be source of troubles and the problem is solved by adding a static IP route toward the message queue via the configuration interface, which is always connected directly to external world.

In conclusion, assuming the scenario shown in Figure 6.2 as initial state before the execution of the present command, after the execution the vNSF is up and running and the IoT Proxy environment configuration becomes the one represented in Figure 6.3.

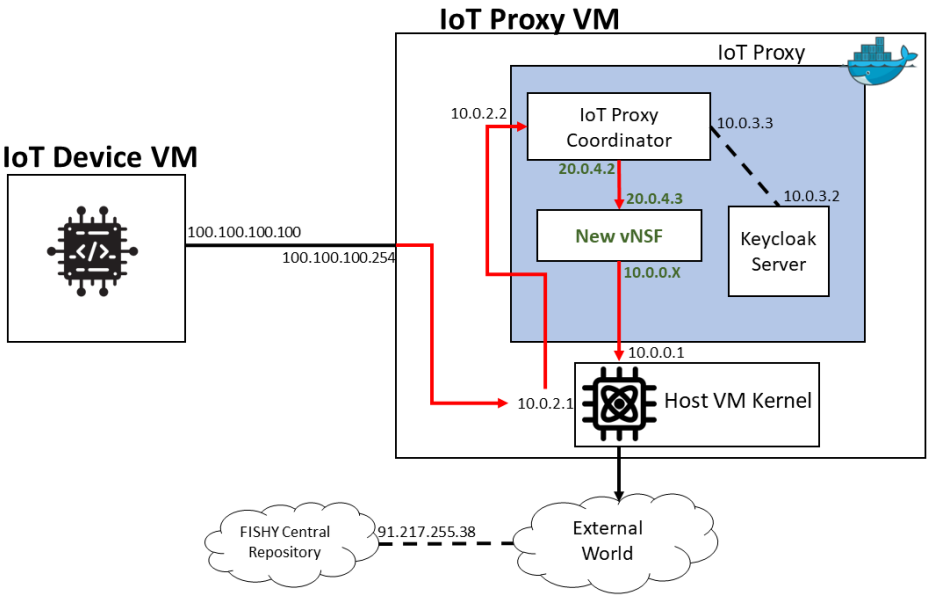

Figure 6.3. IoT Proxy environment configuration after the activation of a new vNSF

vNSF stopping

- API: `/stop_vnsf/<vnsf>/?access_token=<authentication_token>`
- central repository: `STOP_VNSF <vNSF_name>`

This configuration command allows to stop the execution of a previously started vNSF. The API path and the central repository command are built using just one parameter, which is the name associated to the vNSF to stop. As previously specified, during the activation, the name associated to the vNSF was used to name the correspondent Docker container. Subsequently, it is immediate to trace back to the Docker container which has to be stopped.

First, when this configuration command is executed, as usual, IoT Proxy coordinator opens the named pipe shared with the host in write mode and performs correctness checks. In this case, it is checked whether the vNSF to be stopped is actually active or not and if the vNSF is not active, no actions are performed and the coordinator function immediately returns. It is due to cite that, in this context, IoT Proxy coordinator is considered a vNSF as well, but there is the presence of a check which avoid its deletion. Moreover, if the vNSF to stop correspond to strongSwan, the corresponding Docker container is immediately disconnected from the network where the IPsec VPN terminator resides.

At this point, a distinction must be made. In fact, the coordinator must perform a different set of operations, based on whether the vNSF to stop is the last one of the cascade, which is directly connected to the external network, or the aforementioned vNSF is “internal” to the chain, which means it is placed between the coordinator and at least another vNSF in the chain.

If the received command asks for stopping the last vNSF in the chain, first, its bottom interface is disconnected from the external network. Subsequently, the preceding vNSF bottom interface is disconnected from the network which links it with the vNSF to be stopped, and the same interface is connected instead to the external network. Finally, default gateway for the preceding vNSF is replaced with the address of the host system, which as usual in this context means external world, and NAT masquerading functionalities are enabled on it by inserting the corresponding iptables rule. At this point, the preceding vNSF has become the gateway for the whole chain and the Docker network which linked it with the vNSF to be stopped is removed, since it is no more used.

On the other hand, if an "internal" vNSF must be stopped, first its position in the chain and the networks it is connected to are retrieved from IoT Proxy coordinator internal data structures. Consequently, top and bottom interfaces of the interested vNSF are disconnected from the related "top" and "bottom" networks and, immediately after, the preceding vNSF is disconnected from the network which links it with the vNSF to be stopped. Thereafter, the preceding vNSF is connected to the above mentioned "bottom" network and its default gateway is replaced with the address of the next vNSF on that network. At this point, the vNSF to be stopped is completely cut off the cascade and the ex-bottom Docker network is removed, since it is no more used.

In conclusion, in both cases, the Docker container which runs the vNSF to be stopped is killed and then removed. Assuming the initial state of the environment was the one depicted in Figure 6.3, and the configuration command received asks for the stopping of the vNSF represented in green, after the command is executed the environment state becomes the one represented in Figure 6.2.

strongSwan configuration

- API: `/strongswan_addgateway/<username>/<password>/?access_token=<authentication_token>`
- central repository: `SWAN_ADDGW <gateway> <PSK>`

strongSwan is one of the vNSF supported by IoT Proxy version which in delivered in the context of the present work. As aforementioned at the beginning of the present Section, strongSwan will be presented in Section 6.2.2. IoT Proxy provides the possibility to configure the strongSwan vNSF, when it is active.

In particular, the present configuration command provides the possibility to register a new VPN gateway to the one running in the context of IoT Proxy, which is started when strongSwan Docker container is activated, through the methods described above. The API path and the central repository command provide two parameters, which consist of the gateway IP address or certificate common name and the PSK (Pre-Shared Key) to associate to the gateway for which the registration is requested.

When receiving this configuration command, IoT Proxy coordinator opens as always the named pipe shared with the host in write mode and checks that strongSwan vNSF is actually active. Thereafter, extracting username and password parameters values from the API/command, which consists in the certificate common name and the PSK to be associated to this new VPN gateway, it sends through the named pipe a command which indicates to add a new line containing the necessary information to configure the new known gateway, by writing a new line at the end of the `/etc/ipsec.secrets` file within the strongSwan vNSF Docker container.

This file contains a raw for each user which is registered to the VPN, in the following form: `<common_name> : PSK "<PSK>"`. After the execution of this command, the new VPN gateway is able to connect to IoT Proxy in VPN using the configured PSK, in case it is configured to manage accesses in this way.

- API: `/strongswan_config/<config_command>/?access_token=<authentication_token`
- central repository: `SWAN_CONF <generic_conf_command>`

This configuration command allows to send arbitrary commands to the strongSwan vNSF. The API path and the central repository command are built including just one parameter, which consist of a specific shell command to be executed in the context of strongSwan Docker container.

When receiving this configuration command, IoT Proxy coordinator opens the usual named pipe in write mode and checks whether the strongSwan vNSF is active or not. In case strongSwan is active, another check is performed, ensuring that the shell command to be executed is admitted. Admitted commands in this context are the ones indicated in Table 6.4.1.

Through the usage of these commands it is possible to fully configure the strongSwan VPN terminator and, after the execution of the present IoT Proxy configuration command, the requested shell command is executed within the strongSwan container.

Table 6.2. Possible configuration commands for strongSwan vNSF

Command	Description
`pki`	provides a command-line interface allowing clients to access various services, such as certificates, groups, keys, security domains, and users
`ipsec`	command comprising a collection of individual sub commands that can be used to control and monitor IPsec connections
`service`	used to start, stop, and restart the daemons and other services under Linux operating systems
`swanctl`	tool used to configure charon, a daemon built to implement the IKEv2 [72] protocol

6.4.2 Keycloak

The overall IoT Proxy environment provides the presence of a Keycloak server, as shown in Figure 6.2, which is responsible to provide API calls authentication. As mentioned in Section 6.2.1, when IoT Proxy is used as a standalone solution, it exposes a set of API to receive configuration commands and the calls to these APIs must be authenticated to be processed, which means the submission of a valid authentication token is required. In the context of the present work, Keycloak represents the solution adopted for providing API calls authentication.

In the world of IAM[23], Keycloak, an open-source confirmation server written in Java programming language, has emerged as a powerful and popular open-source solution. It is focused on modern applications administration and it makes it simple to protect applications, with almost no code, offering an extended set of features with the purpose of managing user identities, securing applications and enforcing access controls, such as SSO[24] or social login [73].

KeyCloak provides administration and access control for web applications. This means that, using Keycloak, applications do not need anymore to manage all the aspects related to users authentication and authorization on their own. In a sense, it could be said that Keycloak enables an "authentication-as-a-service" paradigm. Moreover, Keycloak supports OAuth 2.0 framework, which represents the industry-standard protocol for authorization, and industry standards for federated authentication, such as OpenID Connect and SAML.

To effectively use and configure Keycloak, it turns out to be essential to understand its core concepts, namely realms, clients, and users. In the context of Keycloak, a realm represents a security domain which manages a set users, applications, and their associated resources, such as credentials and groups. A realm actually contains, isolates and organizes different Keycloak entities.

Each realm has its own set of configurations, authentication flows, and user stores. As an example, consider a software development organization. The creation of different realms makes it possible to enable the separation of identities, authentication mechanisms, and authorization policies between different environments, such as development, testing, and production. Summarizing, realms can be seen as independent zones of trust. Essentially, users and clients within a realm are allowed to interact, while being completely separated from their counterparts who belong to a different realm. In the context of the present work, one single realm was created, namely "iot-proxy-fishy".

[23]Identity and Access Management: the set of technologies, policies and procedures that allows organizations to control users access to applications or data.

[24]Single Sign-On: represents the property of an access control system which allows a user to perform a single valid authentication process to access more systems or resources.

Another Keycloak core concept is represented by clients. A client consists of an application or service which interacts with Keycloak for authentication and authorization purposes. Clients can be represented by any kind of application or service and it must be associated to a realm. When a client is created in a realm, it gains the ability to authenticate the users defined within to the same realm. In the context of the present work, IoT Proxy coordinator application represents the client and one client was created for the purpose, namely "iot-proxy-client". In each client, it is possible to define specific set of configurations such as authentication mechanisms or access policies.

Therefore, clients play a vital role in the overall security architecture of an environment. Clients rely on features offered by Keycloak for identity verification and attribute retrieval of users, as well as access token management, function which turns out to be particularly important in the context of the present work, as discusses at the beginning of the present Section. Summarizing, the definition a client within a realm introduces the possibility to centralize authentication and authorization logic for the application or service associated to the client, ensuring secure access to it.

To effectively benefit from the features offered by Keycloak, the actual application or service must be logically linked to a Keycloak client. In the context of this work, the coordinator, when IoT Proxy is used as a standalone solution, actually consists of a Python Flask application, as said in the first part of the present Section. Flask module supports OpenID Connect framework, which was used to link the coordinator Flask application to the corresponding Keycloak client.

This linkage is realized by means of providing some configuration parameters, such as the realm name, and definition of a `client-secret.json` file, which contains information on where to retrieve access tokens and the client secret. The client secret is a client-specific string generated by Keycloak and associated to the client, which must be provided to associate the application to the specific Keycloak client.

The last presented Keycloak concept is users. In Keycloak, a user represents an individual or in general an entity who interacts with applications secured by Keycloak. Each user is associated with a specific realm and has a set of attributes, including a username, password, email, and optional additional information. Users authenticate with Keycloak to gain access to the applications or services registered within the same realm.

In the context of this work, a single user with an associated password was created, namely "iot-proxy-user". After the creation of the user in "iot-proxy-fishy" realm, it is possible to contact the Keycloak server, providing user credentials, with the purpose of getting an access token, which is represented by a string that contains information about the identity and associated roles or permissions of the user. The aforementioned token can from there on used to authenticate API requests directed to IoT Proxy.

In addition, Keycloak offers others authentication mechanisms for users, such as social logins (e.g. login using Google or Facebook account), and also supports multi-factor authentication[25]. Once authenticated, regardless to the used authentication mechanism, user is issued an access token which can be used for authenticate the user to the applications and services in the context of the same realm.

Finally, it is due to cite that Keycloak user management includes a set of features like user registration, password reset, and account verification, providing to client applications the possibility to almost completely externalize the managing of users authentication and authorization, avoiding the need of designing custom authentication/authorization schemes and therefore likely enhancing the overall security of the application, relying on standard secure mechanisms.

In conclusion, through to the concepts of realms, clients, and users, Keycloak represents a secure and scalable IAM solution. In the context of the present work, Keycloak usage turned out to be fundamental for the security of the presented solution, introducing a form of authentication which makes it possible only to authenticated users to interact with IoT Proxy.

[25] also known as strong authentication, is an authentication method which allow a user to access an application or a system through the submission of two or more proofs.

Chapter 7

Experimental Results

In the context of the present work, two different machines are used to perform the tasks. In particular, a first machine, the technical specifications of which are indicated in Table 7.1, played the role of IoT Proxy testing platform for aspects related to overhead introduced by IoT Proxy. Moreover it was used for the development process of IoT Proxy coordinator and the surrounding vNSFs, as well as the overall Docker environment to test the solution. Moreover, this first machine was used to perform the generation of the features plots used to prune the features showing low correlation.

Operating system	GNU/Linux Ubuntu 20.04.1
Kernel release	5.15.0-72-generic
Python	3.8.10
Docker	20.10.21 (build baeda1f)
docker-compose	1.25.0
scikit-learn	1.1.0
CPU	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
RAM	8 GB

Table 7.1. Characteristics of the IoT Proxy development platform

In addition, a second machine played the role of machine learning training and test platform and was used to perform tasks related to machine learning scope, from datasets building through the analysis of the raw capture files to the machine learning models training and test. Hardware specifications and relevant software products versions of this second machine are represented in Table 7.2.

Operating system	GNU/Linux Ubuntu 20.04.4 LTS
Kernel release	5.4.0-122-generic
Python	3.8.10
scikit-learn	1.1.0
pipenv	2022.5.2
CPU	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
RAM	16 GB

Table 7.2. Characteristics of the machine learning models training and test platform

In the present Chapter, the experimental results obtained in the context of the present work are presented. First, Section 7.1 shows the experimental results obtained in the context of the machine learning component of the IPS vNSF. In particular, within it, Section 7.1.2 reveals the results obtained during the feature selection process, introduced in Section 6.3.1; moreover, Section 7.1.3 and Section 7.1.4 present, respectively, the metrics obtained for the evaluation of the

category classification and the device type classification machine learning models used within the IPS module. Finally, Section 7.2 presents the results obtained in the evaluation of the overhead introduced by the usage of IoT Proxy in the speed of IoT devices network communications.

7.1 Machine learning performance

Machine learning represents a crucial aspect of the proposed solution. In particular, as presented in Chapter 4 and 6, the core of the proposed IPS vNSF integrated in IoT Proxy, implementing also the oblivious authentication idea, is based on machine learning techniques. Two different machine learning models are provided, to carry out two different types of classification tasks, besides the attacks pattern recognition, which consist in the category and device type classification of IoT devices, based on the observation and analysis of the network traffic related to them. As introduced at the beginning of the present Chapter, a machine with the characteristics indicated in Table 7.2 was used for the purpose.

7.1.1 Dataset analysis

As mentioned in Section 6.3, the starting data used for the machine learning part of this work consist in the CIC IoT Dataset 2022. On these data, provided by means of network traffic capture files, `Tstat` tool was used to extract statistics and build the actual datasets with the purpose to train the two proposed machine learning models.

Table 7.3 shows the list of the traffic statistics provided by `Tstat` used in the context of the present analysis, obtained after pruning certain features as a consequence of the considerations presented in Chapter 6 and the feature selection process presented in Section 7.1.2, including a brief explanation of what each metric represents.

The resulting datasets consist each one of 699 262 entries, each one labeled with the associated class, intended as category, device type or attack type. In both cases, the entire dataset is split in two disjointed parts: the 80% of the built dataset, consisting in 559 410 entries was used during the training phase of the machine learning models, while the remaining 20%, consisting in 139 852 entries, was used to perform the test phase, and consequently calculate metrics to evaluate the performance of the model on unseen data.

It is worth remarking one type of analysis performed in the context of category recognition, related to the dataset constitution. This analysis consisted of the calculation of the average value for each available feature, separated by category. This analysis, for which the most relevant results are represented in Table 7.4, is necessary to give an idea of the fact that, for the various categories, the values of the available features are generally different, so that pre-announcing the possibility that the machine learning model will achieve good performances with the available data, since the categories turn out to be distinguishable each other.

In particular, it should be noted that the “Cameras” category, along with the “Other” category, show the highest values for features influenced by the number of exchanged packets. This situation could be caused by the presence of the device “LG TV”, which consists of a smart television, as it will be presented in Section 7.1.4, in the “Other” category, thus representing an element which involves a large amount of traffic, making the average value raise for the related category. However, as it can be noticed from Table 7.4, these two categories show differences related to the traffic direction. In fact, that the average values for the features related to the traffic from server to client are higher for the “Other” category than for the “Cameras”, which instead show the highest values when considering actual bytes sent from client to server, thus adding another useful element of distinction for the machine learning model. These values are due to the nature of the considered IoT devices. In fact, likely, the functioning of an IoT camera provides that it continuously send multimedia data to a central server.

Another reasoning can be made by analyzing the behavior of “Audio” devices. The distinctive element for these devices is represented by a contained amount of traffic sent by the client to the server, while on the opposite direction the average values significantly raise. This is again due to

Statistic	Description
c2s_packets	total number of packets sent by the client
c2s_reset_count	0 = no RST segment has been sent by the client
c2s_ack_pkts	number of segments with the ACK field set to 1 sent by the client
c2s_pureack_pkts	number of segments with ACK field set to 1 and no data sent by the client
c2s_unique_bytes	number of bytes sent in the payload by the client
c2s_data_pkts	number of segments with payload sent by the client
c2s_data_bytes	number of bytes transmitted in the payload by the client, including retransmissions
c2s_rexmit_pkts	number of segments retransmitted by the client
c2s_rexmit_bytes	number of bytes retransmitted by the client
c2s_out_order_pkts	number of segments from the client observed out of sequence
c2s_syn_count	number of SYN segments sent by the client (including rtx)
c2s_fin_count	number of FIN segments sent by the client (including rtx)
s2c_packets	total number of packets sent by the server
s2c_reset_count	0 = no RST segment has been sent by the server
s2c_ack_pkts	number of segments with the ACK field set to 1 sent by the server
s2c_pureack_pkts	number of segments with ACK field set to 1 and no data sent by the server
s2c_unique_bytes	number of bytes sent in the payload by the server
s2c_data_pkts	number of segments with payload sent by the server
s2c_data_bytes	number of bytes transmitted in the payload by the server, including retransmission
s2c_rexmit_pkts	number of segments retransmitted by the server
s2c_rexmit_bytes	number of bytes retransmitted by the server
s2c_syn_count	number of SYN segments sent by the server (including rtx)
s2c_fin_count	number of FIN segments sent by the server (including rtx)
completion_time	Flow duration since first packet to last packet
c2s_payload_start_time	Client first segment with payload since the first flow segment (in ms)
c2s_payload_end_time	Client last segment with payload since the first flow segment (in ms)
c2s_ack_start_time	Client first ACK segment (without SYN) since the first flow segment (in ms)
s2c_payload_start_time	Server first segment with payload since the first flow segment (in ms)
s2c_payload_end_time	Server last segment with payload since the first flow segment (in ms)
s2c_ack_start_time	Server first ACK segment (without SYN) since the first flow segment (in ms)

Table 7.3. Traffic statistics provided by `Tstat`

the nature of the device present in this category, all consisting of vocal assistants. These devices work by sending punctual requests to a central server after having elaborated the vocal command locally. On the other hand, the server, after having received the request and performed all the necessary actions to retrieve the response to the question, sends a potentially big amount of data used to build the vocal answer to the question made by the user.

Finally, additional considerations can be made on the "Other" class. The devices within this category are heterogeneous and encompasses simple devices like smart plugs to more complex ones like smart boards. This heterogeneity could be useful to understand the average values of the features for this category, which, in fact, never represent the maximum or minimum value, for all the features analyzed. However, the traffic belonging to this heterogeneous category can be cleanly distinguished by the machine learning model, since, as all the other categories, it shows

Feature	Category					
	Audio	Cameras	Flood	Home A.	Other	RTSP
c2s_packets	448.981	6168.131	445.853	870.553	6111.847	4.278
c2s_pureack_pkts	350.401	83.658	0.982	405.737	6090.093	1.716
c2s_data_pkts	97.097	6082.953	443.173	463.565	20.733	1.161
c2s_data_bytes	33230.373	1561189.226	642393.973	49572.435	9620.346	251.679
s2c_packets	667.917	5859.921	193.836	740.520	12191.715	3.119
s2c_data_pkts	605.192	92.886	0.902	441.989	12161.956	1.108
s2c_data_bytes	767227.461	8526.009	688.148	35839.536	17018873.562	222.547
s2c_rexmit_pkts	4.485	1.208	0.042	4.648	18.757	0.113

Table 7.4. Average values of most relevant features for the category recognition model

some peculiar characteristics, such as moderately high values for features influenced by the amount of traffic from client to server, which are significantly lower with respect to values for "Cameras" and "Other", but still distinguishable from the lower values showed by the "Audio" category.

The analysis related to the dataset constitution was performed also for the dataset used for device type recognition. This last analysis consisted of the calculation of the average value for each available feature, separated by device type. This analysis results provide evidences of the higher complexity of the device type classification task with respect to category classification. This is highlighted by the fact that, given the more granularity of the classification task and the similarity of the single devices each other, in more cases the features values can be similar for several devices. However, also for this second model, this analysis announces the possibility that the machine learning model will achieve good performances, since, analyzing the results, several elements of distinction can be identified even when considering similar devices. In addition, this deduction is confirmed by the aforementioned results obtained when evaluating the model through the calculation of metrics, as it is shown in Section 7.1.4.

Considering devices belonging to "Audio" category, while these devices share functional similarities, a closer look at their network traffic statistics average values exposes noteworthy distinctions. Metrics such as "c2s_packets," "s2c_packets," and "completion_time" exhibit significant variations among them, as it is shown in Table 7.5. In particular, the results highlighted the fact that the "Google Nest Mini" sends more data to its server with respect to the other devices, while "Amazon Echo Dot" shows an opposite behavior, sending less data but receiving more than the others. The "Sonos One Speaker" element of distinction is represented by the much lower amount of packets exchanged in general. Finally, a different connection completion time for the considered devices can also represent a valid characteristic for classification.

Feature	Device type		
	Amazon Echo Dot	Google Nest Mini	Sonos One Speaker
c2s_packets	368.082	423.292	54.982
s2c_packets	597.222	284.730	45.673
completion_time	2841495848.568	4970511850.036	1181609820.185

Table 7.5. Average values of relevant features for "Audio" devices in category classification model

An interesting result emerges when analyzing devices categorized under "Cameras" category. Despite their functional similarities, elements of distinction can be found, notably in metrics such as "c2s_data_pkts", "c2s_data_bytes", "c2s_payload_start_time" and "c2s_payload_end_time". Specifically, the findings highlight that "Borun Camera" operates distinctively by engaging in shorter connections, witnessed by the proximity of the statistics "c2s_payload_start_time" and "c2s_payload_end_time" average values. The other "Cameras" devices, instead, exhibit a preference for longer-lasting connections characterized by substantial data exchange within a single connection. Some relevant numeric results are shown in Table 7.6.

Another notable revelation comes to the forefront when examining devices classified under the

Feature	Device type			
	Arlo Basestation	ArloQ	Borun	Home Eye
c2s_data_pkts	2324.956	517.044	0.177	96.742
c2s_data_bytes	3321088.026	726573.331	209.541	139353.456
c2s_payload_start_time	106724.606	120732.388	3575.328	206633.037
c2s_payload_end_time	7977621.508	16341112.148	16875.997	859087.921

Table 7.6. Average values of relevant features for "Cameras" devices in category classification model

"Home Automation" category. This category comprises heterogeneous device types which provide really different functionalities, thus facilitating the classification model in the identification of the single device. However, even when considering functionally equivalent devices, such as smart plugs from different vendors, despite their functional uniformity, a closer examination of network traffic characteristics uncovers significant distinction elements, particularly in metrics such as "c2s_data_pkts," "c2s_ack_pkts," "c2s_pureack_pkts," and "completion_time", as shown in Table 7.7. The relatively high number of TCP packets with the "ACK" flag set, typical of all these devices, could be caused by the particular functioning of these devices, which are usually contacted with a specific cadence by a central management system, using a "SYN" TCP packet, to know if they are still active. However, the results highlight differences in the average values calculated over the aforementioned features, that are influenced by the frequency with which the device is contacted. This insights reinforce the idea that a nuanced consideration of network data can unveil valuable distinctions among seemingly similar devices, enhancing the precision of device type classification model when considering devices within the "Home Automation" category.

Feature	Device type			
	Amazon Plug	Gosund Plug	Tekin Plug	Yutron Plug
c2s_packets	172.767	1306.434	292.692	673.096
c2s_ack_pkts	171.668	1305.336	291.568	672.012
c2s_pureack_pkts	120.499	652.607	156.723	338.482
completion_time	3561551810.838	158941871312.803	19730064001.581	80778658409.244

Table 7.7. Average values of relevant features for "Home Automation" devices in category classification model

Lastly, the classification of device types belonging to the "Other" category can be considered the simplest task from the machine learning model point of view. This is because this category comprises device types which are really different each other, showing functional characteristics which are unique in the whole dataset. For this reason, the three devices belonging to this category show differences in almost all the features considered in the present analysis, as it is partially demonstrated by the numeric results in Table 7.8, differentiating themselves for aspects related both to the amount of traffic generated and to temporal aspects or peculiar characteristics of the traffic.

In conclusion, examining the results obtained from this analysis, it can be inferred that the datasets built in the context of this work, starting from the capture files provided by CIC IoT Dataset 2022 and extrapolating aggregated statistics with Tstat, is really likely suitable to train a machine learning model for category classification, since the cases in which more than one category shows values values really similar to the others are rare. This deduction is confirmed by the aforementioned results obtained when evaluating the model through the calculation of metrics, presented in Section 7.1.3 and Section 7.1.4.

7.1.2 Feature selection

As introduced in Section 6.3.1, with respect to the machine learning part of the present work, a features selection process was performed. The data used to train both machine learning models

Feature	Device type		
	DLink Water Sensor	LG TV	Netatmo Weather Station
c2s_packets	251.107	5829.794	2347.288
c2s_pureack_pkts	169.607	5811.492	1729.199
c2s_data_pkts	80.071	17.278	617.053
c2s_data_bytes	30986.071	10484.146	36801.875
s2c_packets	271.893	11638.318	2614.971
s2c_data_pkts	61.857	11624.734	615.902
s2c_data_bytes	3598.179	16270685.027	15031.508
s2c_rexmit_bytes	0.107	25684.672	0.007
completion_time	13234295376.357	78713293.801	186162378309.717
c2s_payload_start_time	115656.5	95021.683	231606.353
s2c_payload_start_time	132747.75	115293.306	229781.734

Table 7.8. Average values of relevant features for "Other" devices in category classification model

are represented by statistics calculated over the network traffic related to the IoT devices. The aforementioned features selection process was performed in order to identify features which show low correlation with the output class, thus turning out to be not particularly useful for the training of the machine learning model, and exclude them from the analysis with the goal to improve the model performances.

The features selection process consisted of the analysis of features values distribution plots, in order to discover which features show low correlation, through the identification of impulse distributions. When the distribution of a given feature manifests an impulse, that feature is considered as not useful, since almost all the entry of the given dataset have the same values for that specific feature, regardless of the class. Therefore, the feature does not provide to the machine learning model a useful element to distinguish classes, but instead introduces a waste of cognitive power. Essentially, the algorithm wastes "cognitive energy", practically translated to computational power, to try to reckon on a feature that is not correlated with one or another output class. Removing this feature, the machine learning algorithm can use this computing power to better analyze the features that are really needed.

Initially, a global plot of each feature was realized, producing graphs which represent the totality of the features values and, after that, the analysis was refined by zooming in on areas where the overall picture of the distribution showed some sort of impulse to discriminate whether indeed each entry had the same value, or the distribution manifested different values but condensed into a given range.

As an example, consider the traffic statistic which represents the number of packets sent from the client to the server, provided by `Tstat`. The first global distribution plot shows an impulse in correspondence of the interval that goes from the value 0 to the value 20 000 approximately, as shown in Figure 7.1. By zooming the picture and tightening the interval width, instead, the distribution does not show a clear impulse, but it can be inferred that the feature assume diverse values, as shown in Figure 7.2, therefore becoming a potentially useful element for the machine learning model training process.

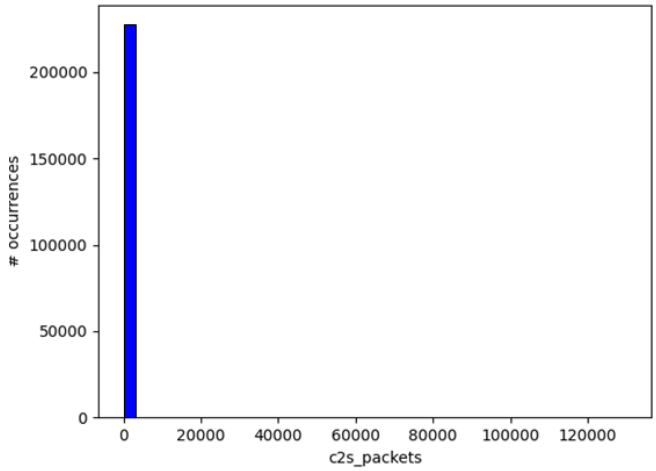

Figure 7.1. Distribution of values for "c2s_packets" feature (not zoomed)

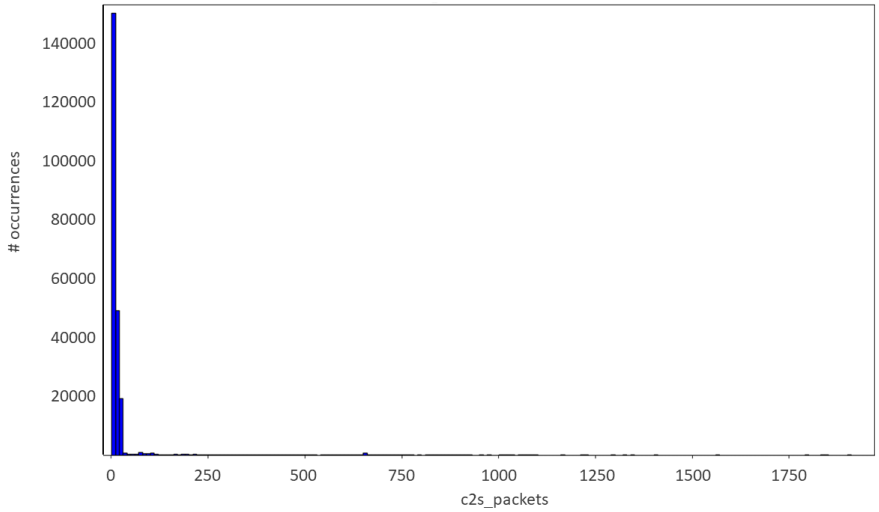

Figure 7.2. Distribution of values for "c2s_packets" feature (zoomed)

Through the features distribution plots analysis, the selection process result consists in the exclusion of only the "s2c_out_order_pkts" traffic statistic. This feature distribution plot shows an impulse, with all the occurrences condensed in correspondence of the value 1, as shown in Figure 7.3. Therefore this feature was excluded since, considering the available data, it does not represent an element of distinction between classes for the machine learning model.

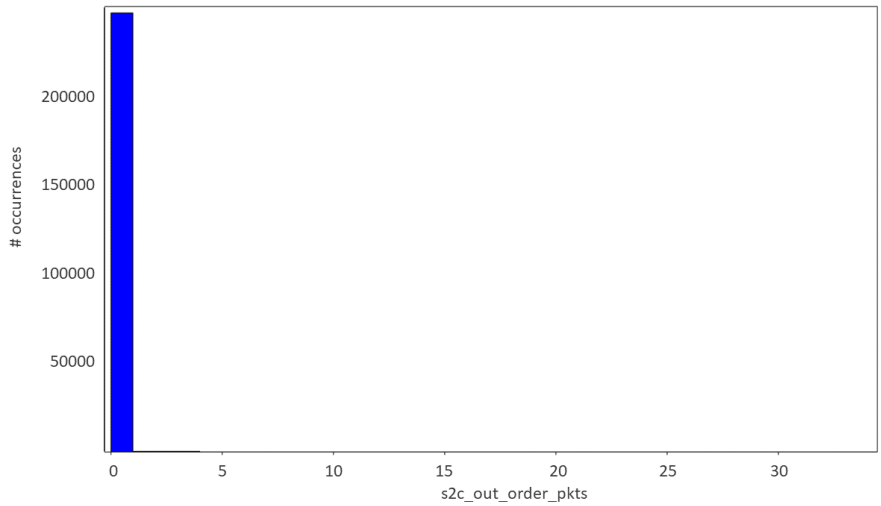

Figure 7.3. Feature distribution of "out of order packets from server to client" statistic of Tstat

7.1.3 Category classification model

With respect to the category classification model embedded in the IPS module of IoT Proxy, it is possible to evaluate its performances through the calculation of several machine learning typical metrics, to show its capacity in recognizing device categories and attacks. The provided classes of the dataset used to train and test the present machine learning model are: "Audio", "Cameras", "Flood", "Home Automation", "Other" and "RTSP Attack". These classes include the possible device categories ("Audio", "Cameras", "Home Automation", "Other") and possible recognizable attacks ("Flood", "RTSP Attack").

In particular, the class "Audio" represents a set of smart speaker devices, while "Cameras" and "Home Automation" classes represent, respectively, smart surveillance cameras and devices used in domotics, like smart lamps or smart vacuum cleaners. Moreover, the class "Other" comprises additional types of devices not falling into the previous categories, such as a smart television and a water sensor. Lastly, the classes "Flood" and "RTSP Attack" refer to recognizable attacks, in particular TCP flooding and RTSP bruteforce attacks, respectively, which have been presented in Chapter 5.

First, the confusion matrix calculated over the test set is shown in Table 7.9.

Actual category	Predicted category					
	Audio	Cameras	Flood	Home Au.	Other	RTSP Att.
Audio	16591	21	5	21	11	229
Cameras	25	11656	11	14	4	306
Flood	3	7	79263	4	0	697
Home Au.	77	15	2	3520	6	41
Other	59	10	1	10	8737	14
RTSP Att.	20	2	0	2	0	6624

Table 7.9. Confusion matrix of category classification model

As demonstrated by the confusion matrix, the model achieves good performances in distinguishing all the various classes. First, it can be noticed that the model performs well in the distinction of traffic statistics related to attacks from normal network traffic. In fact, considering the flooding attack, the model classifies correctly 79 263 test items, representing the 99.1% of the

total test items for the class. Similarly, when considering the RTSP attack, the model correctly classifies 6624 out of 6648 test items, with an accuracy of 99.6% for the specific class.

Moreover, from the confusion matrix it can be deduced that the model shows good performances also in distinguishing the device categories each other in case of non-malicious traffic. In fact, the model performs good predictions for 98.2% of the cases for the audio devices category, for 97% of cases when considering cameras, for 96.1% of times for home automation devices and for 98.9% of cases when considering the "Other" category.

From the confusion matrix it is possible to calculate aggregated metrics over all the available classes, for which the results are shown in Table 7.10, where it can be noticed that the model shows excellent values for all the considered metrics when evaluating unseen data. In particular, the value 0.980 for the balanced accuracy metric symbolizes good capacity by the model in classifying both the minority and majority classes. Moreover, the values obtained for precision and recall, respectively 0.988 and 0.986, indicate good accuracy and completeness of positive predictions, as also demonstrated by the obtained F score of 0.986.

Besides from the calculation of machine learning model evaluation metrics considering the test set, it is possible to calculate the same values for the training set. The scope of this analysis is to understand how the model performs on already seen data and then compare the obtained values with the ones calculated when considering the test set, aiming to detect possible overfitting scenarios. The results obtained when calculating the metrics for the training set are shown in Table 7.10.

Metric	Data considered	
	Test set	Training set
Balanced Accuracy	0.980	0.989
Precision_M	0.988	0.991
Recall_M	0.986	0.989
F score_M	0.986	0.990

Table 7.10. Performance metrics of the category classifier considering test and training set

Comparing the results obtained when considering the test and the training set, indicated in Table 7.10, it is possible to affirm that the proposed model is in the good fit zone. In fact, the results obtained when evaluating the performance using the test set are very slightly lower than the ones obtained when considering the training set, such as for the balanced accuracy that decreases of just 0.009, showing that the model is able to perform well with unseen data, providing good generalization properties.

7.1.4 Device type classification model

With respect to the device type classification model embedded in the IPS module of IoT Proxy, the same kind of analysis presented for the category classification model, discussed in Section 7.1.3, are performed to evaluate its performances. Here, the possible classes provided by dataset used to train and test the present machine learning model are: "Amazon Echo Dot", "Amazon Plug", "Amcrest", "Arlo Basestation Camera", "ArloQ Camera", "Atomi Coffee Maker", "Borun Camera", "DLink Camera", "DLink Water Sensor", "Eufy Homebase", "Flood", "Globe Lamp", "Google Nest Mini", "Gosund Plug", "HeimVision Camera", "HeimVision Lamp", "Home Eye Camera", "LG TV", "Luohe Camera", "Nest Camera", "Netatmo Camera", "Netatmo Weather Station", "Philips Hue Bridge", "RTSP Attack", "Ring Basestation", "Roomba Vacuum", "SimCam", "Smart Board", "Sonos One Speaker", "Tekin Plug" and "Yutron Plug". Each class represents a possible IoT device type or attack type for which the capture files are present in the CIC IoT Dataset 2022.

The presented list of classes represents the list of the device types used in the CIC IoT Dataset 2022. Each device belongs to one of the categories presented in Section 7.1.3. In particular, the smart speakers devices ("Amazon Echo Dot", "Google Nest Mini" and "Sonos One Speaker")

belong to the "Audio" category and consist of devices which are controlled by supplying spoken commands and are capable of replying with audio information. In addition, all the "Cameras" devices ("Arlo Basestation Camera", "ArloQ Camera", "Borun Camera", "DLink Camera", "HeimVision Camera", "Home Eye Camera", "Luohe Camera", "Nest Camera" and "Netatmo Camera") are devices whose purpose is to record video content and transmit it over the network.

Differently, the category "Home Automation" encloses heterogeneous devices. As an example, in this category smart plugs ("Amazon Plug", "Gosund Plug", "Tekin Plug", "Yutron Plug") and smart lamps ("Globe Lamp" and "HeimVision Lamp"), which respectively work as remote control power switches and remote control lamps, as well as a smart coffee maker ("Atomi Coffee Maker"), are present. A smart board ("Smart Board") and a remote controlled vacuum cleaner ("Roomba Vacuum") are also present. Finally, a set of devices which work as central stations ("Eufy Homebase", "Philips Hue Bridge" and "Eufy Homebase") for other sensors from the same vendor complete this category.

Lastly, the category "Other" includes devices which do not fall in any of the aforementioned categories, such as a smart television ("LG TV"), a water sensor ("DLink Water Sensor") which is able to detect water leaks and a weather station ("Netatmo Weather Station"), which is able to provide a series of environmental information like temperature or humidity, are present.

Again, the first task consists of the calculation of the confusion matrix over the test set. To make the confusion matrix of this model understandable, some graphical adjustments are needed because of the large number of columns. For this reason, the resulting confusion matrix is split in three parts and the parts are respectively represented in Table 7.11, Table 7.12 and Table 7.13, while the Table 7.14 plays the role of a legend explaining the codes used for the rows and columns of the aforementioned tables.

As demonstrated by the confusion matrix, also this model manages to achieve good performances in distinguishing almost all the various classes. First, it can be noticed that the model performs slightly worse, but still pretty well, in the distinction of traffic statistics related to attacks from non-malicious network traffic from IoT devices. In fact, considering the flooding attack, the model classifies correctly 79 101 out of 79 908 test items, representing the 98.9%. On the other hand, when considering the RTSP attack for this model, it correctly classifies 5390 out of 6462 test items, showing a lower accuracy for the specific class, precisely 83.4%. As it can be deducted from the confusion matrix, this reduction in the accuracy is given by the fact that the model in some cases misclassifies the traffic related to the RTSP bruteforce attack mistaking it for traffic related to DLink Water Sensor device. However, a pretty good accuracy is still achieved.

Moreover, from the confusion matrix it can be deduced that the model loses a bit of accuracy when considering Amcrest Camera and DLink Camera, sometimes mistaking them each other. In particular, it can be noticed that when considering Amcrest Camera, the traffic related to this device is instead classified as associated with DLink Camera 33.9% of the times, causing an accuracy of 59.2% for the class.

The cause of this misclassiffication could be identified in the very similar nature of these two devices. Being the aforementioned devices both cameras, the behavior and consequently the traffic profile related to these two devices appear to be really similar, therefore introducing an element of confusion for the machine learning classification algorithm. However, when considering DLink Camera, for which the available dataset contains much more data (6328 entries compared with 1333 for Amcrest Camera), the accuracy raises to 81.8%, thus indicating that, with the eventual availability of more data for DLink Camera, the model would probably correct this defect.

An analogue situation verifies when considering Tekin Plug against Yutron Plug. Being these two devices both simple smart plugs, and therefore having a really similar behavior, the machine learning model probably does not have sufficient elements of distinction to efficiently classify them one against the other. However, 70.4% of accuracy is achieved when considering Tekin Plug device, while 76.8% is the value obtained for Yutron Plug, indicating that the model is not extremely precise but it is still above the guessing threshold.

Consequently, from the confusion matrix it is possible to calculate aggregated metrics over all the available classes, for which the results are shown in Table 7.15, where it can be noticed that the model shows pretty good results for all the considered metrics when evaluating unseen

Actual category	Predicted category											
	A	B	C	D	E	F	G	H	I	J	K	L
A	17619	2	20	4	5	0	264	0	276	4	3	0
B	4	63	0	0	0	0	0	0	4	1	0	0
C	28	0	790	6	0	0	452	0	44	0	0	0
D	3	0	13	1942	10	0	3	0	23	0	0	0
E	2	0	0	10	3512	0	0	0	32	0	0	0
F	0	0	0	0	0	58	0	0	7	2	0	0
G	255	0	583	3	1	0	5177	0	265	0	18	0
H	1	0	0	0	0	0	0	64	0	0	0	0
I	0	0	0	0	0	0	0	0	3	0	0	0
J	2	0	0	0	0	0	1	0	2	11	0	0
K	4	0	0	0	0	0	21	0	774	0	79101	0
L	0	4	0	0	0	0	0	0	1	5	0	120
M	19	3	0	0	0	0	0	0	11	1	1	1
N	0	0	0	0	0	1	0	0	15	5	0	6
O	0	0	0	0	0	0	0	0	6	0	0	0
P	4	3	0	0	0	3	1	0	4	3	0	0
Q	0	0	0	1	0	0	0	0	19	0	10	0
R	43	0	0	4	3	0	10	0	45	0	0	0
S	4	0	0	0	0	0	0	0	0	0	0	0
T	1	0	0	0	0	0	0	0	16	0	0	0
U	6	0	1	0	0	0	1	0	16	0	1	0
V	0	0	0	0	0	0	1	0	1	0	0	0
W	2	0	0	4	0	0	3	0	15	0	0	0
X	13	0	0	0	0	0	4	0	1047	0	0	0
Y	2	0	0	0	0	0	1	0	2	3	0	0
Z	5	0	0	0	0	0	0	0	3	0	0	0
AA	6	0	0	0	0	0	2	0	23	0	3	0
BB	36	0	0	1	1	1	3	0	12	1	2	0
CC	34	2	0	0	0	0	0	0	13	2	1	0
DD	2	0	0	0	1	27	0	0	9	0	0	0
EE	0	0	0	0	1	20	2	0	6	5	2	0

Table 7.11. Confusion matrix of device type classification model (1/3)

data. In particular, even if the device classification model perform unsurprisingly worse than the category classification one, due to the higher task granularity and complexity, it still provides acceptable results when evaluating its performances. In particular, the balanced accuracy value 0.849 is significantly lower than the value 0.980 obtained for the other model, decreasing by about 14%. This is due to the fact that, as explained in Chapter 4, the balanced accuracy is a metric that is not influenced by the classes unbalance, which is present in this context, and the model, given the classification granularity that is required for this problem, in some cases misclassify certain device types. On the other hand, the excellent values obtained for precision and recall, and consequently F score, are dependent on the classes unbalance, therefore they provide good prospects but they are not completely reliable in this particular case.

Besides from the calculation of machine learning model evaluation metrics considering the test set, also in this case it is worth calculating metrics considering the training set. Again, the scope of this analysis is to understand how the model performs on already seen data and then compare the obtained values with the ones calculated when considering the test set, aiming to detect possible overfitting scenarios. The results obtained when calculating the metrics for the training set are shown in Table 7.15.

Comparing the results obtained when considering the test and the training set, indicated in Table 7.15, it is possible to notice that the balanced accuracy value significantly decreases when considering the test set. In fact, the value obtained when evaluating the balanced accuracy using

Actual category	Predicted category											
	M	N	O	P	Q	R	S	T	U	V	W	X
A	9	1	1	1	0	22	14	0	1	0	1	19
B	0	2	0	0	0	1	2	0	0	0	0	1
C	0	0	6	0	0	0	7	0	0	0	0	0
D	0	0	3	0	0	1	4	0	0	0	5	0
E	0	0	0	0	0	3	3	0	2	0	0	0
F	0	1	1	2	0	1	0	0	0	0	0	0
G	3	2	0	0	0	8	5	0	1	0	1	0
H	0	0	0	0	0	0	2	0	0	0	0	0
I	0	0	2	0	0	0	1	0	0	0	0	0
J	0	0	1	0	0	1	1	0	0	0	0	0
K	1	1	0	0	1	1	0	0	0	0	0	1
L	0	9	1	1	0	0	1	0	0	0	0	0
M	3290	0	1	0	0	3	2	0	0	0	0	1
N	0	605	4	1	0	1	4	0	0	0	4	0
O	0	0	76	0	0	0	5	0	0	0	0	0
P	0	1	2	169	0	0	2	0	0	0	0	0
Q	0	0	5	0	737	0	5	0	0	0	0	0
R	2	0	0	1	0	9409	0	0	5	0	3	5
S	0	0	0	0	0	0	13	0	0	0	0	0
T	1	0	2	0	0	0	6	1140	0	0	0	0
U	1	0	2	0	0	1	3	0	813	0	0	0
V	0	1	1	0	0	0	0	0	0	83	0	0
W	0	0	0	0	0	1	2	0	0	0	1263	0
X	1	0	0	0	0	3	0	0	0	0	4	5390
Y	0	0	1	0	0	0	0	0	0	0	0	2
Z	0	0	1	0	0	0	1	0	0	0	0	0
AA	0	0	0	0	0	0	7	0	0	0	0	1
BB	7	3	1	1	0	5	3	0	0	0	1	0
CC	2	0	0	0	0	4	3	0	0	0	0	2
DD	0	3	1	2	0	1	2	0	1	0	0	0
EE	0	5	1	4	0	0	0	0	0	0	0	0

Table 7.12. Confusion matrix of device type classification model (2/3)

the test set is lower by about 10% with respect to the one obtained when considering the training set. However, a pretty good absolute value of 0.849 for the balanced accuracy and a overall well-distributed confusion matrix for the test set, help ward off the overfitting hypothesis.

7.2 IoT Proxy overhead

An analysis on the network bit rate, which determines the temporary intensity of a data stream, was performed on the IoT Proxy environment, to provide objective data which show the overhead introduced by the utilization of IoT Proxy. The measurements and analysis are performed using the machine specified in Table 7.1 running `iperf`[1], which allows, among its other features, to automatically measure the maximum achievable bandwidth on IP networks. In the context of the present work, the measurements scenario consists of a virtual IoT device which sends traffic to the default gateway, which means to the machine where IoT Proxy is hosted, eventually passing through IoT Proxy coordinator and modules, which run by means of Docker containers.

[1] https://iperf.fr/ [Accessed: Jul 23, 2023]

Actual category	Predicted category						
	Y	Z	AA	BB	CC	DD	EE
A	1	1	5	6	18	0	0
B	0	0	0	0	1	0	0
C	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0
F	0	0	0	0	0	21	21
G	0	0	1	1	0	2	2
H	0	0	0	0	0	0	0
I	0	0	0	0	0	1	0
J	0	0	0	0	0	0	0
K	0	0	0	1	0	1	1
L	0	0	0	0	0	0	0
M	0	0	2	4	4	0	0
N	0	4	0	0	0	8	5
O	0	0	0	1	0	0	0
P	0	0	0	0	0	1	0
Q	0	0	0	0	0	0	0
R	0	0	0	2	4	5	0
S	0	0	0	0	0	0	0
T	0	0	0	0	0	0	0
U	0	0	0	1	0	0	0
V	0	0	0	0	0	1	0
W	0	1	1	0	2	4	2
X	0	0	0	0	0	0	0
Y	176	0	0	3	0	0	0
Z	0	63	0	0	0	2	0
AA	0	0	826	0	0	0	0
BB	8	0	0	491	0	0	2
CC	1	0	0	1	1029	0	0
DD	0	2	0	0	0	258	57
EE	0	0	0	0	0	61	225

Table 7.13. Confusion matrix of device type classification model (3/3)

The test environment used in this context is represented in Figure 7.4, which represents the most complete test scenario, in which all the supported IoT Proxy modules are activated. The red lines in the picture represent the hops the traffic of IoT devices has to be pass through. In fact, in presence of IoT Proxy, the traffic coming from the virtualized IoT device has to reach the gateway virtual machine, then cross all the Docker containers which run the IoT Proxy coordinator and the various modules, and finally go out to the external world again passing through the gateway virtual machine.

In particular, the traffic coming from IoT device has to pass through the kernel of the machine which hosts IoT Proxy several times, leading to a total of ten forwarding steps which are necessary for the traffic to perform a trip, as shown in Figure 7.4. In details, the traffic is received from the machine which hosts the IoT Proxy from the IoT devices network, then it is forwarder to IoT Proxy, it crosses the whole chain and gets back to the machine. Consequently, to go reach back the IoT device, the traffic traces back the whole IoT Proxy modules chain, reaches the coordinator and then is sent back to the host machine, which forwards it on the actual IoT devices network.

The results of the measurements are represented in Table 7.16, where the obtained bitrates, measured in various configurations, are shown.

Being all the test environment virtualized, the initial bitrate, measured in the absence of IoT Proxy, actually consists of a virtual machine simulating an IoT Device which communicates with

Device Type	Code
Amazon Echo Dot	A
Amazon Plug	B
Amcrest	C
Arlo Basestation Camera	D
ArloQ Camera	E
Atomi Coffee Maker	F
Borun Camera	G
DLink Camera	H
DLink Water Sensor	I
Eufy Homebase	J
Flood	K
Globe Lamp	L
Google Nest Mini	M
Gosund Plug	N
HeimVision Camera	O
HeimVision Lamp	P
Home Eye Camera	Q
LG TV	R
Luohe Camera	S
Nest Camera	T
Netatmo Camera	U
Netatmo Weather Station	V
Philips Hue Bridge	W
RTSP	X
Ring Basestation	Y
Roomba Vacuum	Z
SimCam	AA
Smart Board	BB
Sonos One Speaker	CC
Tekin Plug	DD
Yutron Plug	EE

Table 7.14. Legend for the device type classifier confusion matrix

Metric	Data considered	
	Test set	Training set
Balanced Accuracy	0.849	0.933
Precision_M	0.981	0.995
Recall_M	0.961	0.975
F score_M	0.971	0.985

Table 7.15. Performance metrics of the device type classifier considering test and training set

the default gateway, achieving a value of 2.66 Mbits/s. As it can be deduced from the numeric results, when activating IoT Proxy, which interposes between the IoT device virtual machine and the default gateway machine, the bitrate slightly drops to 2.53 Mbits/s, with a relative decrease of about 5% and an absolute drop of 0.13 Mbits/s. This is due to the fact that, when running IoT Proxy, the network traffic from and to IoT devices is forced to pass through the IoT Proxy coordinator Docker container, thus introducing various passages of the network packets through the kernel of the virtual machine where the IoT Proxy runs.

When switching on IoT Proxy modules, the bitrate further unsurprisingly slightly decreases. It can be observed that, switching on the IPS module, the bitrate value descends to 2.34 Mbits/s, exactly 0.19 Mbits/s less than the value obtained considering only the coordinator, with a relative

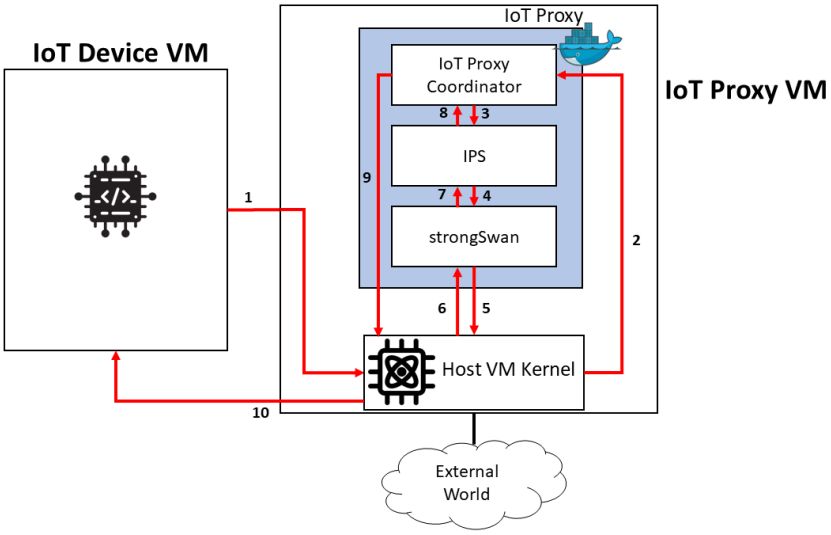

Figure 7.4. Test environment

Table 7.16. Obtained bitrates with various IoT Proxy configurations

No IoT Proxy	2.66 Mbits/s
IoT Proxy coordinator	2.53 Mbits/s
IoT Proxy coordinator + IPS	2.34 Mbits/s
IoT Proxy coordinator + strongSwan	2.33 Mbits/s
IoT Proxy coordinator + IPS + strongSwan	2.20 Mbits/s

loss of about 8%. It has to be mentioned that the same result is obtained when using both the machine learning models, thus indicating that the category and the device type classifiers introduce the same overhead. This is due to the fact that the overhead is only introduce by the presence of an additional Docker container in the chain, since the prediction made by whichever of the two machine learning model is done through the observation of the traffic statistics produced by `Tstat`, which simply sniffs the network traffic transiting on the network interface introducing no overhead in the bitrate.

Similarly, when considering only strongSwan as the active module, the bitrate slightly decreases compared to the scenario in which just the coordinator is active. The measurements are performed without establishing a VPN connection to an external gateway, nor tunneling any traffic, since the overhead introduced by the usage of a VPN is considered as out-of-scope in the context of the present work. Therefore the activation of this module only implies the presence of an additional hop to pass through in network communications between virtual IoT devices and the gateway machine. In particular, it can be observed that, in presence of the strongSwan module, the measured bitrate is 2.33 Mbits/s, showing a value which is really similar to the one measured in presence of the IPS module, thus introducing a really similar relative loss of about 8% with respect to the scenario with no active modules but the IoT Proxy coordinator.

Finally, it can be observed that, activating both the IPS and the strongSwan module, the bitrate is further lowered to 2.26 Mbits/s, introducing a loss of the order of cents with respect to the values observed in absence of one of these two modules. In accordance with the above, the activation of two IoT Proxy modules introduces an overhead which is caused by just the presence of two Docker container in the chain, which consist in two Docker containers the traffic has to be forwarded to. Accordingly, the relative loss introduced by the presence of two modules with respect to one, corresponds to about 6%, while the absolute loss corresponds to 0.14 Mbits/s, which is really similar to the value 0.19 Mbits/s obtained when, in presence of the coordinator, a single module is activated.

Overall, starting from a scenario in which IoT Proxy is not present at all and landing in a situation in which two modules are activated, a bitrate relative loss of about 14% and an absolute loss of 0.35 Mbits/s are observed, introducing a non-negligible but still acceptable overhead. In fact, thanks to the advantages introduced by the usage of a lightweight virtualization technology like Docker, the achieved bitrate in presence of all the possible IoT Proxy modules still shows an high value, not causing discomfort in the network communication for IoT devices and establishing a good trade-off between security and performances.

Chapter 8

Related Works

The proliferation of IoT devices has brought unprecedented opportunities in various fields, from smart homes to industrial automation. However, this widespread interconnection has also exposed IoT devices to a vast amount of security threats. As IoT devices often operate with limited computing power and memory, they become attractive targets for malicious actors who aim to compromise networks and/or gain unauthorized access. Ensuring the protection of these devices and the networks in which they reside is extremely important.

The present work represents a viable solution for the protection of IoT devices networks problem. With respect to the proposed solution, many related works are present in the literature and can be mentioned. The correlation between the present work and the other available in the literature can be indicated by the use of NFV techniques for the protection of IoT environments, the development of ad-hoc IoT-oriented IPS and the usage of VPN solutions in IoT scenarios.

In reference to the concept of using NFV-based solutions for the protection of IoT environments, several works can be found in the literature. Some of them are oriented to malware containment problem and propose solutions based on the virtualization of network security functions in order to tackle the malware propagation in an IoT network [74] [75] [76].

In particular, Aman et al. [74] achieved good results with their proposed security framework that uses security functions virtualization to improve trust in IoT systems and contain the propagation of malware. IoT devices are categorized as trusted, vulnerable and compromised using remote attestation and consequently, according to the category, NFV is used to create separate networks for each category. Their work is focused on 6G, the future and not yet standardized set of technological standards for mobile communications, which among its main features will show the employment of vNSFs. The work provides virtualized remote attestation procedures, defined as "the process of detecting unintentional and malicious modifications to software running on an embedded device by ensuring the integrity of its internal state", in order to prevent potential compatibility issues across heterogeneous IoT devices and efficiently tackle the spread of malware. The outcomes of this work demonstrate that, within a period of 10 seconds, the proposed solution decreases the count of compromised devices by 66%.

Zolotukhin et al. [75] proposed instead a defense system for IoT networks based on SDN, defined by Vmware[1] as an approach to networking that uses software-based controllers or APIs to communicate with underlying hardware infrastructure and direct traffic on a network, and NFV with a core machine learning agent that evaluates risks of potential attack and takes the most optimal action in order to mitigate it. The usage of an SDN model allows the separation of control and data planes and enables the network to be programmable and the underlying infrastructure to be abstracted for applications and network services. The proposed solution provides cloud-based servers, enabling the emulation of real infrastructure components routers and gateways, and the deployment of vNSFs, in particular firewalls, intrusion detection systems, and honeypots. Thanks

[1] https://www.vmware.com/topics/glossary/content/software-defined-networking.html [Accessed: Sep 04, 2023]

to SDN, the system is able to guarantee global visibility of the network state, control the network behavior in a centralized way, and manipulate traffic forwarding rules.

Guizani et al. [76] proposed a software-based architecture that provides NFV capability to combat malware spread for heterogeneous IoT networks, adopting a RNN-LSTM learning model that can predict malware spread for heterogeneous IoT networks in a timely manner, allowing the NFV system to deploy appropriate countermeasures. Aspects related to the scalability of the proposed solution, proved by observing a mild increase in infected nodes with a similar peek infection time as the number of nodes in the network grows, are investigated. The results obtained show that the integration of the proposed machine learning model with a designed patching system that utilizes NFV techniques allows to build a distributed security architecture which is scalable with respect to the size of the network. The ultimate purpose of the work is the development of a generalized IDS, which incorporates the machine learning model and the virtualized patching system, and can deal with a broad range of malwares.

Focusing more on attacks delivered using the network as attack vector in the strict sense, the work from Al-Shaboti et al. [77] proposed a solution consisting of an SDN-based framework for enforcing network static and dynamic access control. This solution integrates an IPv4 ARP server as a vNSF which aims to mitigate ARP spoofing, a common way to perform Man-In-The-Middle (MITM) attacks, by providing ARP replies through a trusted entity and eliminating the usage of ARP broadcast messages. The work mainly focuses on smart home IoT environments and the proposed approach encompasses three key features, which consist of allowing manufacturers to enforce the principle of least privilege for IoT devices, in order to decrease the risks associated with the connection of IoT devices to Internet, facilitating the enforcement of access policies based on feedback from security services, and finally allowing users to customize IoT devices access policies according to contextual requirements, such as restricting the access to IoT devices only to users inside the same local network.

Considering the idea of NFV usage in order to protect IoT environments, the work from Massonet et al. [78] is the most similar to the present one. This work proposes an extended federated cloud networking architecture for edge networks and connected IoT device security. The solution utilizes lightweight virtual network functions and SFC (Service Function Chaining). The IoT gateways are responsible for implementing global security policy, by creating a chain of VNFs for different purposes, such as firewall and intrusion detection/prevention. This vNSFs chain monitors the IoT devices in order to detect vulnerabilities and attacks and isolate the device if a threat is detected, achieved by implementing a federation agent at IoT controller or gateway level. The communication between modules is done using REST API. A federated network manager sends configuration information to IoT network controller, which then forwards these information to agents for actual implementation on IoT devices or gateway level. Finally, the network controller exchanges information with an IoT devices proxy, which is responsible for the data plane management using OpenFlow protocol.

considering the idea of the development of an IoT-oriented IDS/IPS, other several works that show affinity with the solution proposed in this one can be find in the literature. The available solutions differ in the adopted methods. In fact, while several solutions adopt supervised learning techniques [79][49], actually implementing signature-based detection often using Random forest algorithm, others use unsupervised learning techniques [80] adopting a more anomaly-based approach. In addition, other solutions adopt more cutting-edge methods such as blockchain [81] or edge computing [82].

Among the works which adopt supervised learning techniques, S. M. Kasongo [79] proposes an IDS oriented to IIoT applications. The approach followed utilizes the Genetic Algorithm (GA) to perform feature selection, with a random forest model integrated into the GA function, following a so called GA-RF. The intrusion detection models employed encompass a wide range of classifiers, including random forest, decision tree and linear regression. The experimental results obtained show overall good accuracy, reaching values above 85%.

Kumar et al. [49] propose an innovative distributed IDS employing fog computing paradigm to identify and mitigate DDoS attacks within blockchain-enabled IoT networks. Their proposed distributed IDS operates through three key engines. The initial engine, encompasses fog nodes

responsible for preprocessing of the network traffic, performing operations such as feature normalization. Consequently, the second engine focuses on the analysis of incoming IoT traffic adopting two different machine learning algorithms: random forest and XGBoost. Lastly, the third engine, receiving the outcomes from the second one, classifies transactions into either normal or malicious instances and, accordingly, normal transactions are processed while the malicious ones are discarded. To evaluate the efficiency of their proposed model, the authors employed a real-world IoT-based dataset which encompasses the most recent attacks identified within blockchain-enabled IoT networks.

On the other hand, considering solutions following unsupervised learning approach, the work from Basati et al. [80] proposed an innovative network-based IDS based on deep neural networks designed specifically for IoT networks. This NIDS introduces a lightweight architecture based on the Parallel Deep Auto-Encoder (PDAE) deep learning model, which effectively uses information from the immediate local context as well as the broader surroundings of feature values. This approach introduces the possibility to significantly enhance the model accuracy, while contextually reducing the number of parameters, memory usage, and computing power requirements.

In relation to works adopting more modern methods, Sharma et al. [81] proposed a solution based on the usage of a blockchain, defined by IBM[2] as a shared and immutable distributed ledger that facilitates the process of recording transactions and tracking assets in a network, trying to overcome the imitations they identified in centralized security methods. The usage of the blockchain technology offers several advantages, including the elimination of third-party intermediaries, high data integrity and secure peer-to-peer authentication. In this proposed blockchain-based approach, each IoT device is linked to a specific block containing data, a hash, and a timestamp and the data generated by and IoT device is validated through cryptographic hash algorithms provided by the blockchain. The experimental results show that the blockchain integrity validation process performs better than notorious hash algorithms.

Additionally, Jiang et al. [82] work follows the edge computing approach, a distributed model in which data processing takes place as close as possible to where data is generated, improving response time and bandwidth utilization. The proposed solution is based on an edge intelligent gateway, which functions as a lightweight network gateway, deployed on a Raspberry Pi. This gateway provides the execution environment for the detection model and also acts as a Wi-Fi access point for home IoT devices. This gateway employs offers virtual switch capabilities and the customization of modules related to security policies, traffic filtering, and cache management. Despite a resources-constrained device is used as edge gateway, the authors employ a lightweight machine-learning algorithm for the classification of traffic exchanged by IoT devices each other and for the recognition of potential threats. The obtained results demonstrate that this classification model achieves high accuracy in intrusion detection on the edge intelligent gateway while minimizing its impact on network performance.

Because of the special similarity with the proposed solution in the context of the present work, the solution proposed by Goncalves et al. deserves particular mention [83]. Their architectural framework is designed to establish a distributed security system that incorporates the functionality of the Snort IDS for identifying malicious traffic. Additionally, it employs a controller responsible for generating firewall rules on both end hosts and IoT gateways to counteract the detected threats effectively, therefore isolating the infected device from the rest of the network. Furthermore, the architecture includes SDN switches and a central controller entity. When an attack is identified based on its signature database, Snort promptly dispatches an alert to the controller, providing insights about the attack, such as source and destination IP addresses. Leveraging this information, the controller formulates blocking rules to prevent the propagation of the attack and mitigate security events. These security configurations are then transmitted to registered devices to configure corresponding blocking policies also within their iptables.

Another relevant related work in this context is represented by the solution proposed by Haghighi et al. [84]. This work aims to solve the problem related to the false positives predictions of an IPS, presented in Chapter 5. The proposal introduces a machine learning-driven approach

[2] https://www.ibm.com/topics/blockchain [Accessed: Sep 04, 2023]

aimed at constructing an IPS with zero false positives, specifically tailored for IIoT applications. A case study focusing on smart power grids is presented to illustrate its effectiveness. Traditionally, classifiers face the challenge of distinguishing normal traffic from potential attacks based on a set of predefined rules, which may inadvertently filter out legitimate traffic. This becomes especially problematic in critical applications like industrial control systems, where any disruption to normal operations is unacceptable. To address this, the authors have developed an innovative algorithm known as the "z-Classifier", which can enhance the robustness of any generic classifier against false positives. Furthermore, the algorithm's output can be directly translated into firewall rules, particularly when employing a tree-based classifier. It's worth to cite that this approach carries an increased risk of overfitting as its main drawback.

Finally, there are other research efforts similar to the one proposed in the context of the present work, particularly in the realm of solutions that support the usage of VPNs within IoT environments. A review of the literature reveals that a substantial portion of IoT-related solutions opt for VPN utilization. This trend underscores the significance of VPN technology as the primary countermeasure employed in order to provide security to frequently vulnerable IoT network traffic.

In particular, the work from Raj et al. [85] can be cited. The authors designed a prototype of a VPN gateway using a Raspberry Pi, which is then used to connect a home smart IoT devices network with an ISP (Internet Service Provider) network, allowing the IoT devices to benefit from the security services offered by a VPN such as security and scalability. The work born from the skepticism of the authors about the actual security provided by the VPN solutions available on the market, which often collect user data without their explicit consent and show speed limitations due to the allocation of servers to multiple users. Therefore, according to the authors, a private VPN created with a private server located on the cloud and a Raspberry Pi as VPN gateway, following a lightweight approach, is preferred. The experimental results show that the proposed IoT-enabled VPN setup provides about the same performances provided by VPN to the end users in terms of scalability and security.

Analyzing other solutions present in the literature, the result of the research by Fan et al. [86] consists of an IoT gateway built upon the open-source OpenWrt platform, incorporating a comprehensive security mechanism. This gateway adopts an IPSec VPN to safeguard the transport layer of IoT communications. In the context of this work, the authors employ a modular design framework, implementing identity authentication and encrypted communication between the transport and the network layer using China cryptographic algorithm, network security protocols, and PUF (Physical Unclonable Function), physical objects that for a given input and conditions, provide a digital "fingerprint" as output, which can be used as a unique identifier. They also develop the necessary software and hardware components and establish the testing environment. The experimental results indicate that the proposed solution is capable of tackling DoS attacks and spoofing attempts, and showing robust security and stability. The system also offers versatile network access modes and deployment options, effectively fulfilling the demands of data collection, device management, and multimedia converged communication within the IoT environment.

Alharbi et al. [87] proposed the FOCUS (FOg CompUting-based Security) system, which leverages fog computing techniques to offer challenge-response authentication capabilities to IoT devices. The system utilizes a VPN to safeguard the access channel to IoT devices. Furthermore, FOCUS incorporates challenge-response authentication in order to protect the VPN server against distributed DoS attacks, therefore enhancing the overall security of the IoT environment. Essentially, when a client makes its initial connection attempt to the VPN server, it is initially considered as trusted, allowing it to pass through the firewall and connect to the VPN server. However, if a significant volume of network traffic exhibiting similar characteristics is detected (e.g., arriving nearly simultaneously with similar packet sizes, arrival rates, and bursty behavior), FOCUS may label the sources of this traffic as suspicious and subject them to a cryptographic challenge. If the source IP address has been spoofed or if the source is part of a botnet, it will be unable to provide the correct response and, in case the correct answer is not provided, the specific client will then be categorized as untrusted and subsequently blocked by the firewall, denying the access to the VPN server. FOCUS follows a fog computing approach, therefore the data elaboration is performed close to end-users and a fast response time is achieved. The results obtained by the authors during the experiments indicate that the FOCUS system shows a very low response latency and can effectively filter out threats.

Finally, Zedak et al. [88] proposed a secure architecture for remote data acquisition in photovoltaic systems based on IoT sensors for voltage, current, solar irradiation and other measurements. Sensor data is collected and transmitted to an Arduino system, a hardware platform consisting of a series of electronic boards equipped with a microcontroller. Consequently, data are forwarded to a Raspberry Pi which is responsible to store it in the cloud, then allowing users to access this data through a web application. This architecture integrates a VPN terminator utilized by all remote sensors, ensuring end-to-end secure communication and collaboration among them.

The present work aims to improve the protection of IoT devices networks. The proposed solution, thanks to its modular nature, allows easy integration of several security controls and facilities instead of directly focusing on a single technology, therefore allowing the consolidation of several security features in a single product. Moreover, the lightweight virtualization approach followed and the choice of the platform Docker for the deployment guarantees portability.

Chapter 9

Conclusion

The present work has highlighted the urgent need for robust security measures in the context of IoT devices networks. In particular, the interconnected nature of IoT devices has been discussed, posing the focus on design and infrastructural limitations consisting of resource-limited hardware and lack of standardization in this context, added to misconfiguration issues often present trivially because of the neglection during the design, production and deployment stages. The aforementioned security issues typical of IoT devices can be exploited by potential malicious actors in order to compromise the IoT devices themselves, the data they handle and the surrounding environment. In this scenario, FISHY project represents a viable solution in order to tackle these challenges. The project aims to create a coordinated cyber-resilient platform oriented to the establishing of trusted supply chains, which often involve several IoT devices.

The present work consisted in the designing and implementation of the IoT Proxy component of the FISHY platform: a modular secure gateway for IoT devices networks. This solution follows an approach which aims to externalize the management of IoT devices security aspects, providing more robust defense against threats through several security controls implemented in a more powerful centralized system. By channeling all IoT device traffic through IoT Proxy, individual device security is enhanced thanks to the possibility of performing strong security controls, each one consisting of a vNSF.

Thus, the proposed IoT Proxy acts as a first line of defense if integrated within the FISHY platform, since it represents the point of contact between the whole platform and the actual IoT devices. Additionally, even if IoT Proxy born as part of the FISHY platform, the proposed product can also be deployed as a standalone solutions and the security functionalities it offers can be leveraged in order to protect an IoT devices network without the need of the entire surrounding platform.

Practically, the modules of IoT Proxy, representing the provided security controls, have been implemented by means of Docker containers, following an approach which leverages the advantages introduced by the lightweight virtualization paradigm. A central module, implemented as a Docker container as well, is responsible for receiving and performing configurations for the whole IoT Proxy environment.

IoT Proxy incorporates a custom-developed IPS module based on machine learning techniques, capable of identifying and actively mitigating attacks particularly significant in the context of IoT environments. This module also implements the "oblivious authentication" paradigm, further enhancing security detecting anomalous behaviors by IoT devices by analyzing network traffic.

In the context of the IPS module, two machine learning models, respectively oriented to device category classification and device type classification, were trained. In particular, supervised learning techniques were employed and the threat detection problem was modeled as a classification task, in which each attack represents a possible class. The experimental results evidenced good performance for both category and device type classification models, showing high balanced accuracy for both models (98% and 84.9% respectively). In addition, the category classification model was able to detect both types of attacks in more than 99% of cases, while for the other model the values were lower but still high (always more than 83%).

Additionally, IoT Proxy integrates a module which runs strongSwan. The integration of strongSwan, a versatile VPN solution, within the IoT Proxy offers encryption and authentication capabilities to IoT devices, providing the entire IoT Proxy environment with a virtualized VPN gateway which allows, in addition to the possibility of collaboration for IoT devices on different local networks, to secure IoT device communications and safeguarding sensitive data.

Even in scenarios in which both the supported IoT Proxy modules are active simultaneously, the proposed solution shows good results with respect to the network performance degradation introduced. The network connections in these cases are slower but still show acceptable speeds, with a relative loss in performance of about 17%, with each module adding an absolute slowdown of about 0.10 Mbit/s.

The proposed solution shows two main limitations, which are related each other since both are caused by the employment of supervised learning techniques, which will be addressed in future works. The first limitation is represented by the limited number of attacks that the IPS module is able to recognize. Future works will tackle this problem by constructing more complete datasets, including network traffic data related to additional types of even more sophisticated attacks, in order to train the machine learning models with this additional data allowing them the learn to recognize other types of attacks, improving the protection provided by the proposed IPS module of IoT Proxy.

The other limitation of the solution proposed in the context of the present work, always due to the employment of supervised learning techniques were used, is represented by the impossibility in the detection of unknown threats. With this respect, the employment of unsupervised learning techniques, adopting a more anomaly-based approach, is worth investigating in future works.

In particular, one promising avenue can be represented by the usage of autoencoders [89]. Autoencoders represent a class of machine learning unsupervised models, which turn out to be particularly well-suited for anomaly-based detection in network traffic. By training autoencoders on normal IoT device traffic patterns, the system can learn to recognize deviations from these behaviors, thus identifying potential anomalies or intrusions. This anomaly-based approach would offer the advantage of adaptability to evolving threats and the ability to detect previously unseen attacks, introducing the possibility to enhance the ability of the system to identify and mitigate emerging IoT-specific threats.

Finally, another important future direction for the present work consists of the extension of the set of modules supported by IoT Proxy. One interesting possibility towards this direction is represented by the integration of a "security auditing" module, as it was done also for the overall FISHY platform[1]. The idea is to integrate a module that monitors predefined security properties and assesses their implementation, identifying weaknesses and providing insights related to the organization cybersecurity posture querying the other vNSFs.

As an example, the auditing module can interface with the IPS vNSF to retrieve data about the number of triggered alerts or blocked attacks, or it can interact with the strongSwan vNSF in order to know the encryption and authentication algorithms in place. These information will be then employed to generate reports that can then be used to demonstrate that certain levels of security are guaranteed. These reports, automatically generated by the module, can also represent a nice starting point for organizations which plan to acquire certifications for cybersecurity properties, such as the ones issued by ISO[2].

[1] https://fishy-project.eu/blog/role-security-assurance-certification-module-supply-chain [Accessed: Sep 07, 2023]

[2] https://www.iso.org/home.html [Accessed: Sep 07, 2023]

Appendix A

User Manual

A.1 Preliminary information

In order to test and utilize the provided software, the establishment of a dedicated physical or virtual environment is necessary. In this environment, a central machine, equipped with two network interfaces, assumes the critical role of bridging the internal network with the external world. This central machine is the one the IoT Proxy is installed and executed upon and essentially acts as a gateway from the network architecture point of view, enabling communication of the IoT devices connected to the internal network with the external world.

The devices connected to the internal network are real-world IoT devices or virtual counterparts. This setup ensures that the software can effectively monitor and manage the traffic between the internal network and the external environment, offering a controlled and secure testing environment for IoT-related network traffic. Hereinafter, an explanation about how to set up a virtual testing environment, using VirtualBox[1], is provided. For instructions about how to create virtual machines and networks in VirtualBox, as well as how to connect virtual machines to virtual networks, please refer to VirtualBox official documentation (https://www.virtualbox.org/wiki/Documentation).

First, the creation of an Ubuntu virtual machine, preferably with the characteristics indicated in Table 7.15, and equipped with two virtual network cards, is required. The IoT Proxy will be then installed on this machine. After the virtual machine has been created, two network adapters must be activated for it. The settings of the two network adapters are shown in Figure A.1 and Figure A.2. It has to be mentioned that the configuration, including the assignments of IP addresses and the definition of IP routes, has to be done manually inside the single virtual machines for the internal network "iotnet".

Additionally, the following command must be issued on the machine where the IoT Proxy is executed:

```
echo 1 > /proc/sys/net/ipv4/ip\_forward
iptables -P FORWARD ACCEPT
iptables -t nat -A POSTROUTING -o <external interface> -j MASQUERADE
```

Secondly, several additional virtual machines are needed. These machines will act as emulated IoT devices, therefore, they have to be preferably assigned with constrained resources in terms of CPU and memory in order to make the simulation as much realistic as possible. During the testing phase of IoT Proxy, a single CPU core and 2 GB of RAM memory have been assigned to a test machine, and Debian[2] operating system was installed on it. The only virtual network adapter provided for this virtual machine must be connected to the same internal network defined in Figure A.2, as shown in Figure A.3.

[1] https://www.virtualbox.org/ [Accessed: Aug 29, 2023]

[2] https://www.debian.org/index.html [Accessed: Sep 05, 2023]

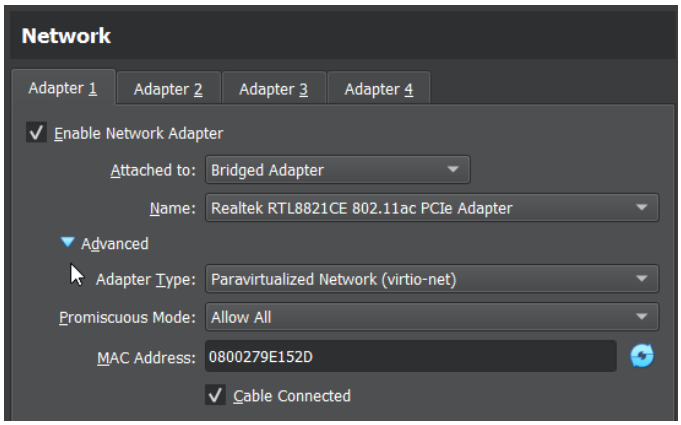

Figure A.1. IoT Proxy virtual machine: first network adapter settings

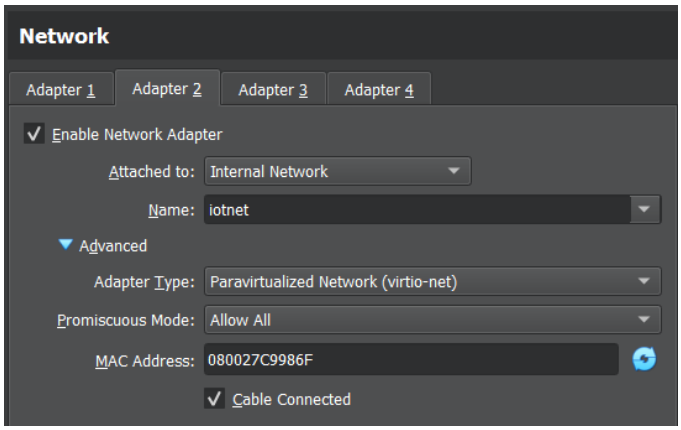

Figure A.2. IoT Proxy virtual machine: second network adapter settings

At this point, all the involved actors are created and the virtual environment is set up from the networking point of view. All traffic related to connected IoT devices will be received by the IoT Proxy VM and is forced to pass through the gateway machine and consequently through IoT Proxy coordinator and any active vNSFs Docker containers, whenever IoT Proxy will be executed.

A.2 Installation instructions

In order to install IoT Proxy, it is first necessary to prepare all the requirements on the host Ubuntu machine. For the sake of compatibility, the `host_requirements.txt` file is provided. This file contains all the packages installed on the Ubuntu machine on which the IoT Proxy has been tested, along with the respective versions. It is due to mention that not all the packages are required in order to install and execute IoT Proxy. Actually, thanks to the adopted lightweight virtualization paradigm, all the components of IoT Proxy are self-contained application, which means that all the requirements they need in order to be executed are enclosed in their Docker images.

Therefore, the only strictly required packages to be installed on the machine where IoT Proxy will be deployed are represented by Docker and docker-compose. For instructions about how to download, install, and use these two tools, please refer to Docker official documentation[3]. After

[3] https://docs.docker.com/ [Accessed: Aug 30, 2023]

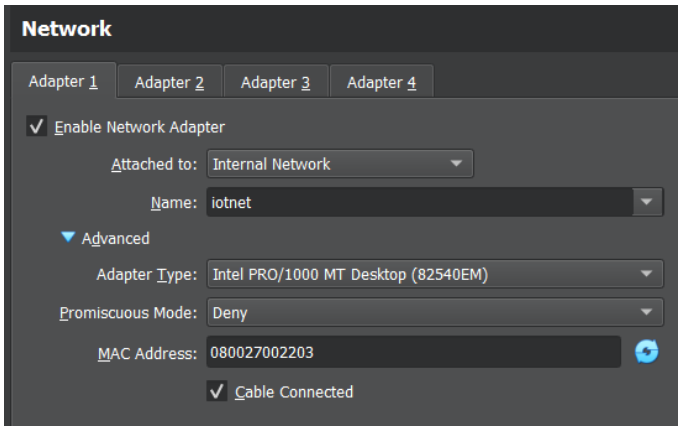

Figure A.3. IoT device virtual machine: network adapter settings

having installed Docker and docker-compose, the system is ready to build all the Docker images of the IoT Proxy components.

A.2.1 Building necessary images

The `docker-compose.yaml` file, used to execute the whole IoT Proxy environment, works with the directive `image` in order to specify the images to use for running the required containers. Therefore, it is necessary to build each individual Docker image and tag them according to the specifications in the `docker-compose.yaml` file. It should be noted that, depending on the configuration on the host, it may sometimes be necessary to run Docker commands with administrative privileges. If so, run all Docker commands using `sudo`.

In order to build all the necessary Docker images, execute the following commands in the project's home directory:

```
docker build -t iot-proxy/iotproxy:0.1 docker-iotproxy
docker build -t iot-proxy/iotproxy:centralrepo -f
    docker-iotproxy/Dockerfile_central_repo docker-iotproxy
docker build -t iot-proxy/obl_auth:device docker-obl_auth
docker build -t strongswan:github docker-strongswan
```

A.2.2 Building Keycloak image

As for the image `iot-proxy/keycloak:20.0.0`, used inside the `docker_compose.yaml` file, there is no available Dockerfile to build from, since the configuration is done starting from a "clean" keycloak image and using the `docker commit` command accordingly, the Keycloak image must be prepared through a set of "manual" steps. The following are the specific steps to configure a ready-to-go Keycloak server with the IoT Proxy, along with instructions for customization if needed. First, run the following command:

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e
    KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/keycloak:20.0.0
    start-dev
```

This will execute a Docker container running a Keycloak server (the Docker image will be automatically downloaded if not already present) listening on port 8080. Consequently, using a web browser, connect to `http://localhost:8080/` and then, on the home page, click on "Administration Console" to access the Keycloak administration panel. Enter the credentials specified in the container execution command:

```
-e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admin
```

In this case, both the username and the password hev been set to "admin".

First, a realm is needed. Once inside the administration panel, use the left-side navigation menu, open the dropdown menu, and select "Create Realm". Fill in the "Realm name" field. Enter "iot-proxy-fishy" for the ready-to-go server. Alternatively, enter your desired Realm name and modify line 30 of the `docker-iotproxy/iot_proxy_rest_server.py` file with the chosen realm name. Ensure that "Enabled" is set to "On" and select "Create".

Next, it is necessary to create a Keycloak Client. Through the Keycloak admin panel menu, select "Clients", and in the displayed screen, choose "Create client". Enter the Client ID = "iot-proxy-client" for the ready-to-go server, or enter your desired "Client ID" and fill in other fields as desired. Continue, enable "Client Authentication", and select "Save".

At this point, at least one user needs to be created. Select "Users" from the navigation menu, then "Add User". Enter "iot-proxy-user" as the username for the ready-to-go server, or enter your desired username, and optionally fill in other fields (not strictly required for functionality), then select "Create". Once the user is created, within the user details, navigate to the "Credentials" tab and select "Set Password". Enter the desired password (e.g., "password" for the ready-to-go server). It is worth to mention that, once set, the password is no longer visible in the admin panel, so make sure to store it securely. Finally, disable "Temporary" and click on "Save".

Navigate to the previously created Client. Select the "Credentials" tab, view the "Client secret", and copy it. If all the instructions for configuring the ready-to-go server have been followed so far, there is only need to replace the copied "Client secret" with the "Client secret" specified on line 6 of the `docker-iotproxy/client-secret.json` file .

Otherwise, if customization were added during the process, some changes are required in the `client-secret.json` file. In such cases:

- replace "iot-proxy-fishy" with your chosen Realm name;
- replace "iot-proxy-client" with your chosen Client name;
- modify "client_secret" on line 6 (as mentioned above).

Finally, open a terminal and run the command `docker ps`in order to display the container identifier of the Docker container running the Keycloak server. Consequently, run the following command:

```
docker commit <CONTAINER\_ID> iot-proxy/keycloak:20.0.0
```

in order to commit the container and create the image. Finally, close the Keycloak administration panel and kill the container, using the command `docker kill <CONTAINER_ID_Keycloak>`. At this point, all the Docker images referenced in the `docker-compose.yaml` file that are necessary for the operation should be ready.

A.3 Usage instructions

Executing IoT Proxy actually means running all the related Docker containers and interconnect them in the proper way. This process is completely performed by means of docker-compose. In this respect, the following items are provided:

- `docker-compose.yaml` file, which defines the container to be executed according to the associated images;
- a `Dockerfile` for each image, necessary to build the related Docker images, as explained in Section A.2.

By default, the `docker-compose.yaml`, when defining the execution template of the IoT Proxy coordinator container, executes the latter in standalone mode, which means the version that exposes REST API in order to receive configurations. If the usage of FISHY central repository is desired instead, you need to:

- Uncomment out lines 10 and 34 of the file `docker-compose.yaml`;
- Comment lines 7 and 32 of the file `docker-compose.yaml`.

At this point, using the command `docker-compose up --force-recreate`, a virtual environment is created as follows:

- IoT Proxy coordinator Docker container: REST server that exposes APIs necessary for configuration and usage, or a message queue client, depending on the chosen mode;
- Keycloak server Docker container: authentication service for API calls.

It has to be noted that, with newer Docker versions, docker-compose has been embedded among Docker commands, therefore, the command to be used becomes the following:

```
docker compose up --force-recreate
```

Note that the presence of clients (real or simulated IoT devices) is not in scope for Docker. In order to test and execute properly IoT Proxy, real or simulated IoT clients which generate network traffic are needed. If IoT Proxy is executed within a virtual machine, the simulated device can consist of an additional VM, connected to an internal network for which the IoT Proxy VM represents the gateway, as explained in Section A.1.

For the functioning, it is necessary for the IoT Proxy coordinator Docker container to communicate the commands to be executed to the host, based on the received configurations. This communication is achieved thanks to a named pipe. Inside the `docker-iotproxy/pipe/` directory, it is necessary to create a named pipe named as `iotproxy-pipe`. Such named pipe is created via the command:

```
mkfifo iotproxy-pipe
```

Consequently, by executing the `execpipe.sh` script provided, the host system listens on this named pipe and executes the received commands. The command to be used is as follows:

```
sudo ./execpipe.sh
```

To stop IoT Proxy execution, simply stop the docker-compose process. Before a new execution, it is recommended to clean-up the whole Docker environment, in order to avoid conflicts with Docker containers and networks names that can cause malfunctioning. The commands to be used are the following:

```
docker kill $(docker ps -a -q)
docker rm $(docker ps -a -q)
docker network prune
```

A.3.1 Usage with FISHY central repository

For IoT Proxy configuration, it is necessary to post the desired configurations to the central repository, through the use of API calls. The calls to be made to post the configuration (one at a time) are of the type:

```
curl https://fishy.xlab.si/tar/api/reports -H 'Content-type:
    application/json' -d '{"source": "test", "data":"<llconf
    nsf=\"iot\_proxy\" format=\"ASCII-text\"
    filename=\"iot\_proxy.config\">COMMAND</llconf>"}'
```

Where `COMMAND` should be replaced, depending on the action you want to communicate to the IoT Proxy, by one of the possible configuration commands presented in Chapter 6. In order to understand the actions that are triggered by the execution of each specific command and for guidance on the vNSFs that can be configured, please refer to the same Chapter 6, where all the various configurations commands are explained. Instead, for the whole list of supported devices and device categories, please refer to Chapter 7, where all the device types and related categories are presented.

A.3.2 Usage with APIs

When using IoT Proxy in standalone mode, configuration commands are sent through REST API requests. In this case, first, it is necessary to authenticate on the Keycloak server obtaining a token necessary for communication with the IoT Proxy, using username and password set during the creation of the Keycloak user, explained in Section A.2.2. The commands to be issued are the following:

```
export USERNAME=iot-proxy-user
export PASSWORD=password
export AUTH\_TOKEN=curl -d "client_id=iot-proxy-client" -d
    "client_secret=<Client_secret>" -d "username=$USERNAME" -d
    "password=$PASSWORD" -d "grant_type=password"
    "http://10.0.3.2:8080/realms/iot-proxy-fishy/protocol/openid-connect/token"
    | jq -r ".access_token"
```

The AUTH_TOKEN received allows the use of the IoT Proxy APIs. Then, through the following command, using the `curl` tool, it is possible to contact the exposed APIs and configure the IoT Proxy:

```
curl -s http://10.0.2.2:5000/<API_path>/?access_token=$AUTH_TOKEN
```

Where `<API_path>` is replaced by one of the possible API endpoints presented in Chapter 6. In order to understand the actions that are triggered by the execution of each specific command and for guidance on the vNSFs that can be configured, please refer to the same Chapter 6, where all the various configurations commands are explained. Instead, for the whole list of supported devices and device categories, please refer to Chapter 7, where all the device types and related categories are presented.

It should be noted that, when used in this mode, IoT Proxy reads its initial configuration from a file named `iot_proxy.config`, where a series of command is specified using the central repository commands format. By default, the configuration file indicates to activate both IPS and strongSwan module. If you want to change this behavior, you can edit the `iot_proxy.config` file and rebuild the IoT Proxy coordinator Docker image, as specified in Section A.2.1.

A.3.3 IPS module usage

The IPS module allows for attacks (Flooding and RTSP bruteforce) prevention. Moreover, it allows network traffic authentication: analysis of traffic statistics related to IoT devices and, through a trained ML model, disconnection of the device from the network (through firewall rules configured using iptables) if the traffic profile deviates from what is expected for the category assigned to the device in question.

In order to actually run the process responsible for traffic prevention, it is necessary to spawn a shell inside the Docker container running the IPS module. The commands are the following:

1. check the ID of the IPS module container through the command `docker ps`;
2. run a shell inside the container through `docker exec -it <IPS_containerID> /bin/bash`;
3. inside the container, run the following command:

```
python3.9 obl_auth.py <interface> <reaction>
    <category/device>
```

The first parameter allows to specify on which interface the traffic has to be sniffed. The possible reactions to be passed as the second parameter, applied when a device that does not comply with its traffic profile, behaving as a different device, are as follows:

- `block`: provides for blocking the connection for the device that does not comply with its traffic profile, behaving as a different device;
- `alert`: prints an alert on the screen when a device does not comply with its traffic profile.

The reaction when an attack is recognized is always `block`, which indicated the isolation of the attack source from the network. The last parameter indicates if the traffic analysis should be done for the single device types or on the related category.

A.3.4 strongSwan module usage

The container running Strongswan comes complete with: certification authority certificate and key, server certificate and key, `/etc/ipsec.conf` configuration file, and `/etc/ipsec.secrets` secrets file. An example PSK-authenticated connection is provided as already configured.

Note that it is necessary to configure the resources that should be reachable by the client once connected in VPN. Specifically, this is indicated in the `leftsubnet` field of the `/etc/ipsec.conf` file on the server and in the `rightsubnet` field of the `/etc/ipsec.conf` file on the client. The server is represented by IoT Proxy strongSwan instance, while the client is any other strongSwan instance who potentially want to connect to it.

It is recommended to configure these fields once the container is run, modifying the files according to the particular situation. In order to do so, spawn a shell in the strongSwan Docker container and manually edit the aforementioned files at need. For any additional information about VPN configuration and operation, please refer to the official Strongswan documentation[4].

Once the container is executed, it is necessary to start the `ipsec` service. This can be done by sending a generic strongSwan configuration command via API or Central Repository, as presented in Chapter 6. Alternatively, this can be done directly from a bash executed in the context of the container. In this case, the commands are the following:

1. check the ID of the strongSwan module container through the command `docker ps`;
2. spawn a shell inside the container through `docker exec -it <strongSwan_containerID> /bin/bash`;

In both cases, the necessary command to be issued is the following:

```
ipsec start
```

If on-screen logging is desired, the `--nofork` flag needs to be appended to the above command (useful only if the service has been started from a shell inside the container). Next, on the client, copy the CA certificate created in the server. The certificate can be retrieved issuing the following command on the host system:

```
sudo docker cp <Strongswan\_Container\_ID>:/etc/ipsec.d/cacerts/ca-cert.pem
    /etc/ipsec.d/cacerts/ca-cert.pem
```

[4] https://docs.strongswan.org/docs/5.9/index.html [Accessed: Sep 04, 2023]

Consequently, the certificate has to be communicated to the other VPN gateways. For this last point, the exact procedure depends on the specific situation. In order to actually use the VPN, sample files `ipsec-client.conf` and `ipsec-client.secrets`, provided in the `docker-strongswan/asymmetric_authentication` folder, can be installed on the client in order to connect. Finally, on the client, run `sudo ipsec start` and then `sudo ipsec up net-net` (net-net is the name of the native connection configured) in order to connect to the IoT Proxy strongSwan instance.

Appendix B

Developer Manual

B.1 Machine learning model training

An explanation of the machine learning model training process is provided. The provided IPS vNSF relies on two random forest classifiers, both of which are dedicated to the task of attacks detection. However, these models serve additional purposes beyond attack detection. In fact, the first model extends its capabilities to classify the specific category to which the detected traffic belongs, while the second model is designed to perform also device type classification. Both models are trained using network traffic statistics, extracted analyzing the sniffed network traffic with `tstat` library. The models used by the IPS vNSF have been trained using the `.pcap` files provided by the CIC IoT Dataset 2022. The provided model has been trained using traffic statistics extracted from the `.pcap` files contained in the "1-Power", "2-Idle", "3-Interactions" and "5-Active" directories.

The utility `find_all_pcaps.py` script is provided, which, given the dataset's location of capture files, extracts the complete list of `.pcap`files. This list is then read by the `build_dataset.py` script, which analyzes the traffic content of each file in the list and extracts the statistics, placing them in separate dataframes along with their corresponding labels. Once the above script is executed, with the labeled dataset of traffic statistics available, the `train_model.py` script can be used to train a Random Forest and save the trained model to a file. This trained model file is then loaded by the IPS script.

At this point, it should be noted that for the purpose of explaining the functioning of the code, the category classification task is considered. The functioning is exactly the same for the device type classification dataset preparation, with the only difference represented by the label assignment function, which label traffic statistics dataframe using device types instead of categories.

The `build_dataset.py` Python script is used to build a labeled traffic statistics dataset from a list of `pcap` files. The list of the files to be taken into consideration is contained in the `PCAPS_LIST` variable, stored within the `constants` Python module, that is discussed, along with its related content, in Section B.2.

Overall, this script's execution flow involves reading a list of `pcap` filenames, analyzing each file to extract traffic statistics and assign labels, and then storing this information in values and labels files. This script is part of a machine learning pipeline for creating a labeled dataset from `pcap` files. Therefore, the dataset is categorized, by adding a label to each dataframe resulting from each `pcap` file analysis, into different IoT device categories based on traffic patterns and it is stored in a text file. This file is later used for machine learning model training and evaluation. The following `build_dataset` function represents the core of the dataset building process:

```
1 def build_dataset():
2     # read line by line pcap list
3     first = True
```

```
4     with open(constants.PCAPS_LIST) as fp:
5         # read a single filename from the list of pcaps
6         pcap_filename = fp.readline().strip('\n')
7         with open(constants.DATAFRAME_VALUES, constants.OPEN_MODE_DF_FILES)
              as values_fd:
8             while pcap_filename:
9                 cp = ConfigParser()
10                cp.read(constants.NETGEN_CONF_FILE)
11                ta = TstatAnalyzer(cp, 50)
12                # find label for that specific path
13                lab = find_label(pcap_filename)
14                if lab == "NoCat":
15                    print("Error analyzing file: " + pcap_filename)
16                    exit("File " + pcap_filename + " is not labellable from its
                          own path!")
17                else:
18                    print("Label for file: " + pcap_filename + " is = " + lab)
19                # analyze a single pcap file
20                try:
21                    s = ta.analyze(pcap_filename)
22                except Exception:
23                    print("Exception during pcap file analysis")
24                    raise
25                for df in s:
26                    df['label'] = lab
27                    if first == True:
28                        df.to_csv(values_fd, index=False, header=True)
29                        first = False
30                    else:
31                        df.to_csv(values_fd, index=False, header=False)
32                pcap_filename = fp.readline().strip('\n')
```

In particular, this function operates by opening a file containing a list of `pcap` files and an output file. Consequently, the function consists of a loop which iterates over all the available `pcap` files available within the list and, for each one of them:

1. row 9-11: instantiate a `TstatAnalyzer` object, using as parameters:
   - a `ConfigParser` object, which is configured to read the `netgen` Python library configuration file, which specifies where all the specific files for the system library;
   - an integer number consisting of the number of packets after which the `Tstat` output dataframe must be returned.
2. row 13: find the correct label to be associated with the dataframes resulting from the file analysis, trivially scanning the file path name and looking for a matching label in the string;
3. row 21: analyzes the `pcap` file using the `analyze` function of the `TstatAnalyzer` object, whose output consists of a sequence of dataframes;
4. row 25-31: for each dataframe in the output sequence, the previously identified label is appended and, consequently, the dataframe is stored in a file using `csv` format.

Consequently, once the `csv` file with all the available data is ready, it is used by `train_model.py` Python script, which is responsible to retrieve the data and use it to train a machine learning random forest classification model. Trivially, this script instantiate a `RandomForestClassifier` object, which is the implementation of a random forest model provided by the `sklearn` library. `class_weight="balanced"` parameter is important since it mitigates the impact of the presence of unbalanced classes on the model performances. Consequently, it retrieves the previously built

dataset from the file using the `get_dataset` function provided by the `build_dataset` module, which simply reads the `csv` file and returns data and associated labels in separated Python lists.

These data and labels are then used to generate the training set, represented by `X_train` and `y_train` variables, and the test set, represented by `X_test` and `y_test` variables. The test set is then used to train the random forest classifier using the `fit` function provided by the `RandomForestClassifier` object, on row 19. After, the test set is used to test the trained machine learning model and for the calculation of several evaluation metrics. Finally, the trained random forest model is stored in a file using the `dump` function from the `joblib` library. This file will be then loaded by the IPS module of the proposed IoT Proxy, as it is discussed in Section B.2.

In order to change the used machine learning model, it is necessary to edit the `train_model.py` script, specifically on line 9, where the desired model is loaded. The `sklearn` library offers several implementations of machine learning models. For a comprehensive view, please refer to the official `sklearn` documentation[1]. It should be noted that, if an unsupervised model is chosen, thus completely changing the logic of the proposed solution, several other adjustments in the scripts interested in the machine learning model training and usage, as well as dataset construction, may be needed.

To change the data used in the machine learning training process, the `build_dataset.py` script and/or the related files most be modified accordingly. In particular, to add or remove `pcap` files interested by the traffic statistics calculation, rows must be added or deleted from the `pcaps.list` file, which is mapped in the `constants.PCAPS_LIST` variable by the `constants` Python module. Instead, if the willing is to calculate statistics over traffic directly captured from the network, there is only the need to edit line 21 of the `build_dataset` function of the `build_dataset.py` script, using the `sniff` function of the `TstatAnalyzer` object instead of `analyze`, specifying the specific network interface to sniff on as the only parameter.

There could be cases in which it could be needed to change the traffic statistics used to train the machine learning model, for example adding a new feature not provided by `Tstat`. In this case, add the code to calculate the additional statistics and append the result as a column in each dataframe, on line 26 of the `build_dataset` function of the `build_dataset.py` script, as it is done for the label. Instead, to modify the `Tstat` set of features used for the analysis, edit the `COLUMNS_FOR_ANALYSIS_OPTIMIZED` list within the `constants.py` module.

If additional categories of IoT devices or device types must be introduced, it is necessary to edit the `find_label` function within the `build_dataset.py` script. The related code is illustrated below:

```
1  def find_label(pcap_filename):
2      if pcap_filename.__contains__("Audio"):
3          return "Audio"
4      if pcap_filename.__contains__("Cameras"):
5          return "Cameras"
6      if pcap_filename.__contains__("Home Automation"):
7          return "Home Automation"
8      if pcap_filename.__contains__("Other"):
9          return "Other"
10     return "NoCat"
```

It should be noted that, to maintain the structure of the function as a simple series of `if` statements, it is necessary that the `pcap` files related to network traffic for the newly introduced devices are stored on a file system location which contains the category and the device type within the absolute path. Otherwise, ad-hoc code must be developed.

Finally, if additional evaluation metrics are desired in order to better understand the performance of the machine learning model, the final part of the `train_model.py` script, from line 23

[1] https://scikit-learn.org/stable/ [Accessed: Sep 20, 2023]

to line 37, must be modified. Again, please refer to the official `sklearn` documentation for all the available evaluation metrics with can be used with each specific machine learning model.

B.2 Code structure and main functions

B.2.1 docker-compose.yaml

The provided `docker-compose.yaml` file is used to define and configure Docker services and networks for a containerized application. The configuration focuses on the definition of two main services, namely `iotproxy` and `keycloak`, and the related characteristics. Finally, a section about the networks configuration is present. For a comprehensive view about `docker-compose` utilization and functioning, please refer to the official `docker-compose` documentation[2].

In particular, the `iotproxy` service represents the definition of the IoT Proxy coordinator Docker container, providing details about its characteristics such as privileges and IP addresses in correspondence of its various network interfaces. The full definition of `iotproxy` service is provided below:

```
1  iotproxy:
2      # Using APIs
3      image: iot-proxy/iotproxy:0.1
4      container_name: iot-proxy
5      # Using central Repository FISHY
6      # image: iot-proxy/iotproxy:centralrepo
7      hostname: iotproxy
8      stdin_open: true
9      tty: true
10     networks:
11         # eth0
12         conf_net:
13             ipv4_address: 10.0.2.2
14         # eth1
15         ext_net:
16             ipv4_address: 10.0.0.2
17         # eth2
18         keycloak_net:
19             ipv4_address: 10.0.3.3
20     privileged: true
21     volumes:
22         - ./docker-iotproxy/pipe:/iotproxy-pipe
23     depends_on:
24        - keycloak
25     cap_add:
26         - net_admin
27     # Using APIs
28     command: sh -c "sysctl net.ipv4.ip_forward=1 && ip r replace default via
           10.0.0.1 && iptables -t nat -A POSTROUTING -j MASQUERADE && python3.9
           iot_proxy_rest_server.py"
29     # Using central Repository FISHY
30     # command: sh -c "sysctl net.ipv4.ip_forward=1 && ip r replace default
           via 10.0.0.1 && ip r add 91.217.255.38 via 10.0.2.1 && iptables -t
           nat -A POSTROUTING -j MASQUERADE && python3.9
           iot_proxy_central_repo.py"
```

[2] https://docs.docker.com/compose/ [Accessed: Sep 20, 2023]

In particular, the `iotproxy` service definition:

- specifies which Docker image to use in order to deploy the Docker container;
- sets the container name as "iot-proxy";
- configures network settings for "conf_net," "ext_net," and "keycloak_net" networks, whose specification are provided below;
- runs in privileged mode, in order to have the possibility to interact with the network card of the host machine;
- mounts a volume from the host machine which will be used to interact with the named pipe on the host machine;
- depends on the `keycloak` service, since the IoT Proxy coordinator needs the presence of the Keycloak server for API calls authentication.
- specifies to run the Python script which implements the IoT Proxy coordinator inside the container.

The `keycloak` service represents the definition of the Keycloak server Docker container, providing details about the characteristics of the container, such as environment variables values and the IP address to use on the network the container is connected to. The full definition of `keycloak` service is provided below:

```
1  keycloak:
2      image: iot-proxy/keycloak:20.0.0
3      container_name: keycloak
4      hostname: keycloak
5      stdin_open: true
6      tty: true
7      ports:
8          - "8080:8080"
9      environment:
10         - KEYCLOAK_ADMIN=admin
11         - KEYCLOAK_ADMIN_PASSWORD=admin
12     networks:
13         # eth0
14         keycloak_net:
15             ipv4_address: 10.0.3.2
16     command: start-dev
```

In particular, the `keycloak` service definition provides the following directives:

- use the `iot-proxy/keycloak:20.0.0` Docker image to deploy the Docker container;
- set the container name as "keycloak";
- expose port 8080 for external access, in order to make the Keycloak server reachable from outside the host machine;
- set environment variables for Keycloak administration credentials;
- configure network settings for the `keycloak_net` network, whose characteristics are described below.

The last part of the `docker-compose.yaml` file represents the definition of characteristics for the Docker virtual networks that are created in order to interconnect the various Docker container that compose the whole IoT Proxy environment. The full definition of the networks is provided below:

```
networks:
    ext_net:
        internal: false
        name: ext_net
        ipam:
            config:
                - subnet: "10.0.0.0/24"
    conf_net:
        internal: false
        name: conf_net
        ipam:
            config:
                - subnet: "10.0.2.0/24"
    keycloak_net:
        internal: false
        name: keycloak_net
        ipam:
            config:
                - subnet: "10.0.3.0/24"
```

In particular, three different Docker networks, serving to three different purposes, are defined:

- `ext_net`: is used for communication between the IoT Proxy outermost module (the last vNSF in the chain) and the host machine, used for IoT devices traffic forwarding;
- `conf_net`: is used in order to connect IoT Proxy coordinator with the host machine, in order to allow sending configuration command to IoT Proxy coordinator;
- `keycloak_net`: is used in order to connect the IoT Proxy coordinator with the Keycloak server.

B.2.2 iot_proxy_rest_server.py

As mentioned in Section B.2.1, after the whole IoT Proxy environment is built by means of `docker-compose`, the Python script which implements the IoT Proxy coordinator is executed inside the `iot-proxy` container. Assuming that the IoT Proxy is deployed in the (default) standalone mode, the coordinator consists of a Python Flask web server that receives configurations through a set of APIs. The server is implemented by means of the `iot_proxy_rest_server.py` script. First, some configuration parameter are specified for the web server, in order to connect the application with the deployed Keycloak server.

Consequently, the set of available APIs is defined, each one by means of the `@app.route`, according to the logic of the Flask framework. The presence of the Flask framework directive `@oidc.accept_token(require_token=True)` in correspondence of each API indicates that the authentication token is required in order to proceed with the execution of the actual associated function. Each API associated function actually represents a wrapper what forwards the parameters received through the API call to the actual function that implements the actions to be performed. The `config_*` functions are defined in the `config.py` Python module, which will be later presented.

Finally, the script provides, through the `if __name__ == '__main__':` conditional statement, some preliminary actions to be performed before the server is started. In particular, a firewall rule which indicated to drop all the traffic to be forwarded is set, representing the basis for the implementation of a whitelist approach. Furthermore, an initial configuration file `iot_proxy.config` that can contain some preliminary configuration commands for the IoT Proxy coordinator is read, and the corresponding actions are executed. Finally, the Flask server is actually started, listening on port TCP 5000. The following code snippets represents the entrypoint for the `iot_proxy_rest_server.py` script execution:

```
if __name__ == '__main__':
    os.system(f"iptables -P FORWARD DROP")
    read_config_file()
    app.run(host="10.0.2.2", port=5000, use_reloader=False)
```

The following script represents a sample `iot_proxy.config` file:

```
        ADD_IOT_DEV 100.100.100.100 ArloQCamera Cameras
        START_VNSF strongswan
        START_VNSF IPS
```

The file consists of a series of configuration commands defined with the central repository command format and handled using the same `read_config` function of `iot_proxy_central_repo.py` Python module, which instead implements the coordinator when IoT Proxy is deployed as part of the FISHY platform, as it will be discussed here below.

B.2.3 iot_proxy_central_repo.py

The `iot_proxy_central_repo.py` script contains the definition of a `RMQsubscriber` class, which provides standard functions and data structure needed to communicate with a remote message queue, in this case represented by the FISHY central repository, implementing a message queue consumer. The class provides the `on_message_callback` function, whose execution is triggered when a new message is posted on the message queue. This function, after checking if the message is intended for IoT Proxy, extracts the configuration command and invokes the `read_config` function, which parses the configuration command string in order to retrieve the actual command to be executed and the associated parameters, which are then passed to the corresponding function of the `config.py` module, actually implementing the actions to be executed. An extract of the function, related to the `START_VNSF` command, is provided in the below code snippet:

```
def read_config(str):
    print(str)
    strings = str.split(' ')
    command = strings[0]
    if command == "START_VNSF":
        vnsf_name = strings[1]
        print("executing start vnsf")
        config_activate_vnsf(vnsf_name)
```

When the `iot_proxy_central_repo.py` script execution is triggered, some configuration preliminary actions are performed before starting the actual message queue consumer, as reported in the following code snippet:

```
queueName = 'FISHYQueue'
key = 'reports.#'
notification_consumer_config = { 'host': 'fishymq.xlab.si', 'port': 45672,
    'exchange' : 'tasks', 'login':'tubs', 'password':'sbut'}
if __name__ == '__main__':
    # Read the Message Queue of the FISHY central repository
    os.system(f"ip r add 91.217.255.38 via 10.0.2.1\n")
    os.system(f"iptables -P FORWARD DROP")
    print("Sent command to add static route to FISHY Central Repository")
    try:
       init_rabbit = RMQsubscriber(queueName, key,
           notification_consumer_config)
       init_rabbit.setup()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
```

```
16          except SystemExit:
17              os._exit(0)
```

In particular, first, some configuration parameters are set in order to connect to the FISHY central repository remote message queue. Thus, the rows 1-3 of the above code snippet represent the portion of code to be modified if a different message queue is desired. In addition, a static IP route to the FISHY central repository, via the host machine, is configured, for the reasons already presented in Chapter 6. Consequently, a firewall rule which indicated to drop all the traffic to be forwarded is set, representing the basis for the implementation of a whitelist approach. Finally, the actual message queue consumer is started, by instantiating a `RMQsubscriber` object using the previously defined parameters.

B.2.4 config.py

The `config.py` Python file contains the actual implementation of the IoT Proxy coordinator logic. As presented in Section B.2.2 and Section B.2.3, after having received the configuration command through an API call or a message posted on the FISHY remote message queue, the IoT Proxy coordinator invokes the corresponding function of the `config.py` module. As already discussed in Chapter 6, two main types of operations are performed in correspondence of a given command, respectively related to Docker containers and networking management. Below, details about each function is provided, focusing on the main functionalities and abstracting away some irrelevant or trivially understandable details in this context, such as error checking on parameters, printing, comments or state variables increasing. All the functions follow the same structure and provide the construction of a command that is then written to the named pipe, when actions need to be performed by the host machine, or executed in the context of the container operating system.

config_activate_vnsf(...)

The `config_activate_vnsf` function is responsible for the actual vNSFs instantiation and activation. First, a new "intra-vNSFs" Docker network is created, via the `incr` variable that allows to always create networks with different names adding a progressive number in the name. Consequently, the behavior of the function is different whether the `IPS` or the `strongSwan` IoT Proxy module must be activated.

In particular, when the `IPS` module activation is required, the function performs the following actions:

- starts the IPS Docker container with specific networking settings, attaching it to the "intra-vNSFs" Docker network and giving it a name based on the value of `vnsf` received parameter;
- configures the iptables firewall inside the IPS container. Specifically, it sets the default policy for `FORWARD` chain to `DROP`, effectively blocking all traffic;
- iterates over items in the device_category_dict, where each item represents an IoT device registered to IoT Proxy. In particular, for each IoT device:
  - extracts the category and device type information from the dictionary;
  - appends device configuration data to a file named `obl_auth.config` within the IPS container, which represents the IPS module configuration file;
  - modifies iptables rules within the IPS Docker container, allowing traffic to and from the specified IoT device IP address.

All the constructed commands are, one by one, written to the named pipe shared with the host system for actual execution.

On the other hand, when the `strongswan` module activation is required, the function performs the following actions:

- creates a Docker network named "strongswan_conf_net" with specific settings, that will be used for actual VPN traffic;
- starts a Docker container named "strongswan" with specific networking configuration that IPsec and then sleeps for a long time (effectively running indefinitely), and attaches it to the "strongswan_conf_net" and to the "intra-vNSFs" Docker network;
- modifies several iptables rules, the ones containing the words `isakmp` and `ipsec-nat-t`, in order to delete the Docker default configuration of ports exposing and to configure the VPN service to be provided through the "strongswan_conf_net"-attached network interface.

Lastly, several actions are performed by the function in order to correctly configure the networking settings for the whole environment. In particular:

- disconnects the penultimate vNSF from the external network and connects it to the new "intra-vNSFs" network;
- connects the newly activated vNSF to the external network;
- loops trough all the active IoT devices and, for each one of them, updates the IP routes on the host machine;
- updates the default route for the penultimate vNSF making the newly started vNSF become the default gateway for it;
- adds IP routes to each IoT devices to the newly started vNSF;
- disables NAT on the penultimate vNSF and enables NAT on the newly started vNSF.

config_register_device(...)

The `config_register_device` function is responsible for the registration of a new IoT device to IoT Proxy. After some initial checks related to the matching of the received device type and category, networking-related configurations are performed. In particular, the function:

- whitelists the newly registered IoT device;
- adds IP routes to the newly added IoT devices in each active vNSFs;
- updates the policy-based routing tables adding the routes to the newly registered IoT device and from it the external world, in the host machine;
- adds an IP route to the newly registered IoT device within the coordinator Docker container.

Lastly, if active, the IPS module is updated, configuring iptables rules in order to allow traffic forwarding to and from the newly registered IoT device, thus adding it to the whitelist, and inserting a new line in its configuration file, in order to make the IPS know the device category and type.

config_remove_device(...)

The `config_remove_device` function is responsible for the removal of an IoT device previously registered to IoT Proxy. Essentially, it consists in a rollback of the operations performed during the registration phase.

config_stop_vnsf(...)

The `config_stop_vnsf` function is responsible for stopping a running IoT Proxy vNSF. The behavior of the function is different according to whether the vNSF to be stopped is the last one in the chain, directly connected to the host machine, or if it is an "internal" one, since the actions to be taken are different.

In case the vNSF to stop is the last one in the chain, during the function execution:

- the last vNSF is disconnected from the external network and the penultimate one is connected to it, disconnecting the penultimate vNSF from the previously configured intra-vNSFs network;
- the host machine is set as default gateway for the penultimate vNSF;
- NAT functionalities are activated for the penultimate vNSF;
- the routes to the IoT devices are updated in the host machine, configuring the penultimate vNSF as the default gateway for the corresponding policy-based routing table.

On the other hand, if the vNSF to be stopped is in the middle of the chain, the function performs the following actions:

- identifies the position of the vNSF to be stopped in the chain and the related Docker networks;
- disconnects the vNSF to be stopped from the two intra-vNSFs Docker networks it was connected to and disconnects the preceding vNSF from the Docker network that connected the two vNSFs;
- connects the preceding vNSF to the intra-vNSFs Docker network that linked the vNSF to delete with its subsequent, and accordingly updates the default gateway for the preceding vNSF.

At the end, the no more necessary Docker container and network are deleted.

config_strongswan_gateway(...)

The `config_strongswan_gateway` function is responsible for adding a new known VPN gateway for the strongSwan vNSF. Essentially, this is performed by appending a new row to the `/etc/ipsec.secrets` file of the strongSwan Docker container using the received parameters. For a comprehensive view of strongSwan functioning and configuration, please refer to the official `strongSwan` documentation[3].

config_strongswan_conf(...)

The `config_strongswan_conf` function is responsible allows to send a generic configuration command to the strongSwan vNSF. Essentially, this is performed by executing the received command string in the context of the strongSwan Docker container, after checking that the command is allowed. For a comprehensive view of strongSwan functioning and configuration, please refer to the official `strongSwan` documentation[4].

[3] https://www.strongswan.org/documentation.html [Accessed: Sep 20, 2023]

[4] https://www.strongswan.org/documentation.html [Accessed: Sep 20, 2023]

B.2.5 obl_auth.py

The `obl_auth.py` Python script is the one that actually implements the intrusion prevention functionalities. The first part of the script is responsible for parameters correctness checking, configuration of the correct machine learning model to be used, and `TstatAnalyzer` object instantiation. Consequently, the core is represented by an infinite loop.

This first part of the loop is in charge of reading the configuration file in order to detect configuration changes, such as the presence of a new registered device. Consequently, the traffic is sniffed and the prediction by the machine learning model is collected, verifying if the predicted value represents the presence of a threat or not, and checking if it matches the actual one for the specific device category or type.

In particular, the script:

- reads the configuration file, updating the configuration, starts the actual sniffing and resets the counters variables;
- once the sniffing provides data, predicts the class for each provided dataframe of the sniffed traffic;
- sequentially checks if the prediction identified an ongoing attack (prediction is "Flood" or "RTSP") and, if not, checks if the prediction output matches the category or device type provided for the related IoT device, whether the IoT device is the client or the server in the communication, and finally updating associated counters variables accordingly.

Consequently, the last part of the loop consists of the implementation of the traffic control logic, according to the values of the predictions. In particular:

- checks if the prediction for the sniffed traffic indicates the presence of an ongoing attack and, if so, removes the attack source from the whitelist, deleting the correspondent iptables rules;
- if the machine learning models does not detect an attack, it checks if the prediction by the model matches the actual category or device type of the interested IoT device, taking the appropriate countermeasures, removing the IoT device from the whitelist (using the same technique described when presenting `config_remove_device` function) or raising an alert according to the configuration;

B.3 Possible integration and extensions

The proposed solution offers several outlets for eventual integration. IoT Proxy born as an extensible and modular solution which allows to easily integrate new functionalities. The main extension is represented by the integration of additional security controls.

In order to introduce an additional vNSF it is first necessary to create a Docker image associated with it. This Docker image must be based on a Linux distribution and have the `iproute2` package installed, since, as shown in Section B.2, the IP routes update and all the other networking-related operations necessary for the functioning of the whole IoT Proxy environment are performed by means of the commands provided by this package.

Given these prerequisites, the Docker image name must be appended to the `name_image_dict` dictionary defined at line 19 of the `config.py` file. This Python dictionary specifies all the supported vNSFs. Therefore, to integrate an additional vNSF, an entry must be added to this dictionary, specifying:

- key: a short name for the vNSF, to be used in API calls or message queue commands;
- value: the name of the Docker image.

Once the entry is added to the aforementioned data structure, the vNSF is fully supported by IoT Proxy, obviously after the IoT Proxy coordinator Docker image is re-built. If the introduced vNSF is a "generic" one and does not require any customization, this is the only operation that needs to be performed.

On the other hand, if the vNSF requires dynamic configuration according to the state of the surrounding IoT Proxy environments, such as the proposed IPS vNSF that requires a configuration file to be updated when new IoT devices are registered, additional operations are needed. In these cases, the necessary operations must be specified within an `if` block inside the `config_activate_vnsf` function of the `config.py` file.

As it is done for IPS and strongSwan vNSFs, the eventual operations needed to configure the new vNSF must be coded within its associated `if` block, where there is also the possibility to interact with the host system through the shared named pipe. If needed, custom cleanup operations associated to the specific vNSF must be specified in the `config_stop_vnsf` function, adding an appropriate `if` block where necessary operations are specified. These represent the only actions needed to integrate an additional security module in IoT Proxy.

Furthermore, there is the possibility to easily integrate additional IoT devices categories and types. This operation is pretty trivial, since the IoT Proxy functioning is agnostic from the connected devices type. The first action needed in order to introduce support for additional categories or device types is to add a list at the beginning of the `config.py` file, as it was done for the already supported devices:

```
audio_devices = ["AmazonEchoDot", ... , "SonosOneSpeaker"]
camera_devices = ["Amcrest", ... , "SimCam"]
home_device = ["AmazonPlug", ... , "YutronPlug"]
other_devices = ["DLinkWaterSensor", ... ,"LGTV"]
```

In particular, the list contains all the device types belonging to the specific category. The correspondence between the new device type and the associated category must be checked inside the `config_register_device` function, in order to avoid mismatches. Obviously, to make the provided IPS module fully support the introduced categories and device types, the machine learning models must be trained accordingly, following the instructions provided in Section B.1.

Lastly, another extension possibility is represented by the integration of additional APIs or configuration commands. In order to introduce an additional API, it is necessary to modify the `iot_proxy_rest_server.py` file, introducing an additional function tagged with the `@app.route` annotation from the Flask framework, specifying the API endpoint. Moreover, if eventual calls to this API endpoint must be validated by means of a Keycloak authentication token, as it happens for the already present APIs, the `@oidc.accept_token(require_token=True)` is required. For a comprehensive view about the functionalities offered by the Flask framework, please refer to the official `Flask` documentation[5].

On the other hand, in order to introduce an additional configuration command to be used when IoT Proxy coordinator is essentially deployed as a message queue consumer, the changes must be introduced in `iot_proxy_central_repo.py` file. If the newly introduced command specification matches the structure convention of the already supported one, consisting of textual strings containing the command followed by the list of parameters separated by a space (i.e., `ADD_IOT_DEV 100.100.100.100 ArloQCamera Cameras`), the only action required consists of adding an `if` block inside the `read_config` function, responsible for the introduced command handling. Otherwise, it is also necessary to develop a custom command parser in order to extract the parameters values.

It should be noted that if the introduced configuration command must be supported also by the API server version of IoT Proxy, when reading its initial configuration file, the same changes introduced in `iot_proxy_central_repo.py` must be written in the `read_config_file` function of `iot_proxy_rest_server.py`.

[5] https://flask.palletsprojects.com/en/2.3.x/ [Accessed: Sep 20, 2023]

Finally, to maintain the code structure and modularity, the functions defined in the files `iot_proxy_rest_server.py` and `iot_proxy_central_repo.py` should consist of simple wrappers invoking an associated function defined in another Python module. In fact, the approach followed in the development of the present solution provides that functions defined inside these files must only serve to extract the parameters values from the received API call or configuration command, then invoking the correspondent function defined in `config.py` were the actual implementation is specified.

Bibliography

[1] M. Lombardi, F. Pascale, and D. Santaniello, "Internet of Things: A General Overview between Architectures, Protocols and Applications", Information, vol. 12, February 2021, DOI 10.3390/info12020087

[2] J. Wurm, K. Hoang, O. Arias, A.-R. Sadeghi, and Y. Jin, "Security analysis on consumer and industrial IoT devices", 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), Macao (Macao), January 25-28, 2016, pp. 519–524, DOI 10.1109/aspdac.2016.7428064

[3] N. Doraswamy, K. R. Glenn, and R. L. Thayer, "IP Security Document Roadmap", RFC 2411, November 1998, DOI 10.17487/RFC2411

[4] ETSI, "ETSI GS NFV 003 V1.2.1: Network Functions Virtualisation (NFV)", December 2014, https://www.etsi.org/deliver/etsi_gs/nfv/001_099/003/01.02.01_60/gs_nfv003v010201p.pdf

[5] P. Mockapetris, "DNS encoding of network names and other types", RFC 1101, April 1989, DOI 10.17487/RFC1101

[6] M. Holdrege and P. Srisuresh, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, August 1999, DOI 10.17487/RFC2663

[7] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges", IEEE Communications Surveys & Tutorials, vol. 18, no. 1, 2016, pp. 236–262, DOI 10.1109/comst.2015.2477041

[8] ETSI, "NFV Whitepaper: Network Function Virtualization, issue 1", SDN and OpenFlow World Congress, Darmstadt (Germany), October 22-24, 2012. https://portal.etsi.org/nfv/nfv_white_paper.pdf

[9] M. D. Benedictis, A. Lioy, and P. Smiraglia, "Container-based design of a Virtual Network Security Function", 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), Montreal (Canada), June 25-29, 2018, pp. 55–63, DOI 10.1109/netsoft.2018.8459903

[10] E. C. S. O. of the European Union., "Eurostat regional yearbook: 2021 edition.", Publications Office, 2021. https://ec.europa.eu/eurostat/documents/15234730/15241943/KS-HA-21-001-EN-N.pdf/202462e1-b947-a2c5-6da2-c3d999018134?t=1667396809671

[11] ETSI, "ETSI GS NFV 002 v1.1.1 - Architectural Framework", October 2013, https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf

[12] T. Li, D. Farinacci, S. P. Hanks, D. Meyer, and P. S. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000, DOI 10.17487/RFC2784

[13] ETSI, "ETSI GS NFV-INF 001 V1.1.1 (2015-01): Network Functions Virtualisation (NFV); Infrastructure Overview", January 2015, https://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_NFV-INF001v010101p.pdf

[14] ETSI, "ETSI GS NFV-SWA 001 V1.1.1: Virtual Network Functions Architecture", December 2014, https://www.etsi.org/deliver/etsi_gs/NFV-SWA/001_099/001/01.01.01_60/gs_NFV-SWA001v010101p.pdf

[15] ETSI, "ETSI GS NFV-MAN 001 V1.1.1 - Network Functions Virtualisation (NFV); Management and Orchestration", December 2014, https://www.etsi.org/deliver/etsi_gs/nfv-man/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf

[16] K. I. Farkas, C. Narayanaswami, and J. Nieh, "Guest Editors' Introduction: Virtual Machines", IEEE Pervasive Computing, vol. 8, October 2009, pp. 6–7, DOI 10.1109/mprv.2009.74

[17] I. Tuomi, "The Lives and Death of Moore's Law", First Monday, vol. 7, November 2002, DOI 10.5210/fm.v7i11.1000

[18] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer", ACM SIGARCH Computer Architecture News, vol. 35, no. 2, 2007, pp. 13–23, DOI 10.1145/1273440.1250665

[19] J. P. Buzen and U. O. Gagliardi, "The evolution of virtual machine architecture", Proceedings of the June 4-8, 1973, national computer conference and exposition on - AFIPS '73, New York (New York, USA), June 4-8, 1973, pp. 291–299, DOI 10.1145/1499586.1499667

[20] I. Corporation., "Iapx 286 Programmer's Reference Manual Including the Iapx 286 Numeric Supplement", Intel Books, November 1984, ISBN: 9780835930543

[21] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends", Computer, vol. 38, May 2005, pp. 39–47, DOI 10.1109/mc.2005.176

[22] D. Sitaram, "Moving to the cloud developing apps in the new world of cloud computing", Syngress, December 2011, ISBN: 9781597497251

[23] B. Rodrigues, F. Cerveira, R. Barbosa, and J. Bernardino, "Virtualization: Past and Present Challenges", Proceedings of the 13th International Conference on Software Technologies, Porto (Portugal), July 26-28, 2018, pp. 755–761, DOI 10.5220/0006910707550761

[24] M. Probst, A. Krall, and B. Scholz, "Register liveness analysis for optimizing dynamic binary translation", Ninth Working Conference on Reverse Engineering, 2002 Proceedings, Richmond (Virginia, USA), November 29-December 01, 2002, pp. 35–44, DOI 10.1109/wcre.2002.1173062

[25] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization", ACM SIGOPS Operating Systems Review, vol. 40, October 2006, pp. 2–13, DOI 10.1145/1168917.1168860

[26] S. L. Harris and D. Harris, "Memory and I/O Systems", Digital Design and Computer Architecture: RISC-V Edition, pp. 499–541, October 2021, DOI 10.1016/b978-0-12-820064-3.00008-8

[27] C. Waldspurger and M. Rosenblum, "I/O virtualization", Communications of the ACM, vol. 55, January 2012, pp. 66–73, DOI 10.1145/2063176.2063194

[28] A. M. Joy, "Performance comparison between Linux containers and virtual machines", 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad (India), March 19-20, 2015, pp. 342–346, DOI 10.1109/icacea.2015.7164727

[29] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers", IBM Journal of Research and Development, vol. 3, July 1959, pp. 210–229, DOI 10.1147/rd.33.0210

[30] P. Cunningham, M. Cord, and S. J. Delany, "Supervised Learning", Machine Learning Techniques for Multimedia, vol. 1, pp. 21–49, DOI 10.1007/978-3-540-75171-7_2

[31] H.B. Barlow, "Unsupervised Learning", Neural Computation, vol. 1, September 1989, pp. 295–311, DOI 10.1162/neco.1989.1.3.295

[32] W. Koehrsen, "Overfitting vs. underfitting: A complete example", Towards Data Science, 2018, pp. 1–12. https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765 [Accessed: Jun 03, 2023]

[33] T. Hastie, R. Tibshirani, and J. Friedman, "The Elements of Statistical Learning", Springer New York, 2009, ISBN: 978-0-387-84858-7

[34] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, "Feature Selection", ACM Computing Surveys, vol. 50, December 2017, pp. 1–45, DOI 10.1145/3136625

[35] A. HAY, "The derivation of global estimates from a confusion matrix", International Journal of Remote Sensing, vol. 9, August 1988, pp. 1395–1398, DOI 10.1080/01431168808954945

[36] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks", Information Processing & Management, vol. 45, July 2009, pp. 427–437, DOI 10.1016/j.ipm.2009.03.002

[37] B. de Ville, "Decision trees", Wiley Interdisciplinary Reviews: Computational Statistics, vol. 5, October 2013, pp. 448–455, DOI 10.1002/wics.1278

[38] C. Kingsford and S. L. Salzberg, "What are decision trees?", Nature Biotechnology, vol. 26, September 2008, pp. 1011–1013, DOI 10.1038/nbt0908-1011

[39] J. R. Quinlan, "C4.5: Programs for Machine Learning", Elsevier Science & Technology Books, 2014, ISBN: 9780080500584

[40] L. Breiman, "Random Forests", Machine Learning, vol. 45, October 2001, pp. 5–32, DOI 10.1023/a:1010933404324

[41] T. K. Ho, "Random decision forests", Proceedings of 3rd International Conference on Document Analysis and Recognition, Montreal (Canada), August 14-16, 1995, pp. 278–282, DOI 10.1109/icdar.1995.598994

[42] D. Canavese, L. Regano, C. Basile, G. Ciravegna, and A. Lioy, "Encryption-agnostic classifiers of traffic originators and their application to anomaly detection", Computers & Electrical Engineering, vol. 97, January 2022, pp. 1–11, DOI 10.1016/j.compeleceng.2021.107621

[43] N. Mishra and S. Pandya, "Internet of Things Applications, Security Challenges, Attacks, Intrusion Detection, and Future Visions: A Systematic Review", IEEE Access, vol. 9, 2021, pp. 59353–59377, DOI 10.1109/access.2021.3073408

[44] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges", Cybersecurity, vol. 2, July 2019, DOI 10.1186/s42400-019-0038-7

[45] Z. Chiba, N. Abghour, K. Moussaid, O. Lifandali, and R. Kinta, "A Deep Study of Novel Intrusion Detection Systems and Intrusion Prevention Systems for Internet of Things Networks", Procedia Computer Science, vol. 210, October 2022, pp. 94–103, DOI 10.1016/j.procs.2022.10.124

[46] E. E. Abdallah, W. Eleisah, and A. F. Otoom, "Intrusion Detection Systems using Supervised Machine Learning Techniques: A survey", Procedia Computer Science, vol. 201, March 2022, pp. 205–212, DOI 10.1016/j.procs.2022.03.029

[47] C. Sinclair, L. Pierce, and S. Matzner, "An application of machine learning to network intrusion detection", Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99), Phoenix (Arizona, USA), December 06-10, 1999, pp. 371–377, DOI 10.1109/csac.1999.816048

[48] S. U. Jan, S. Ahmed, V. Shakhov, and I. Koo, "Toward a lightweight intrusion detection system for the internet of things", IEEE Access, vol. 7, March 2019, pp. 42450–42471, DOI 10.1109/access.2019.2907965

[49] A. Kumar, K. Abhishek, M. Ghalib, A. Shankar, and X. Cheng, "Intrusion detection and prevention system for an IoT environment", Digital Communications and Networks, vol. 8, August 2022, pp. 540–551, DOI 10.1016/j.dcan.2022.05.027

[50] P. Spadaccino and F. Cuomo, "Intrusion Detection Systems for IoT: opportunities and challenges offered by Edge Computing and Machine Learning", ITU Journal on Future and Evolving Technologies (ITU J-FET), December 2020, DOI 10.48550/arXiv.2012.01174

[51] C. M. Lonvick and T. Ylonen, "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006, DOI 10.17487/RFC4253

[52] R. Venkateswaran, "Virtual private networks", IEEE Potentials, vol. 20, February 2001, pp. 11–15, DOI 10.1109/45.913204

[53] S. Khanvilkar and A. Khokhar, "Virtual private networks: an overview with performance evaluation", IEEE Communications Magazine, vol. 42, October 2004, pp. 146–154, DOI 10.1109/mcom.2004.1341273

[54] R. R. Jadhav and P. S. Sheth, "VPN: Overview and Security Risks", International Journal of Advanced Research in Science, Communication and Technology, July 2021, pp. 305–309, DOI 10.48175/ijarsct-1649

[55] E. Rescorla and T. Dierks, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008, DOI 10.17487/RFC5246

[56] R. Atkinson and S. Kent, "Security Architecture for the Internet Protocol", RFC 2401, November 1998, DOI 10.17487/RFC2401

[57] R. Roman, J. Zhou, and J. Lopez, "On the features and challenges of security and privacy in distributed internet of things", Computer Networks, vol. 57, July 2013, pp. 2266–2279, DOI 10.1016/j.comnet.2012.12.018

[58] A. Rubens, C. Rigney, S. Willens, and W. A. Simpson, "Remote Authentication Dial In User Service (RADIUS)", RFC 2865, June 2000, DOI 10.17487/RFC2865

[59] Y. Seralathan, T. T. Oh, S. Jadhav, J. Myers, J. P. Jeong, Y. H. Kim, and J. N. Kim, "IoT security vulnerability: A case study of a Web camera", 2018 20th International Conference on Advanced Communication Technology (ICACT), Gangwon (South Korea), February 11-14, 2018, pp. 172–177, DOI 10.23919/icact.2018.8323686

[60] Information Sciences Institute of University of Southern California, "Transmission Control Protocol", RFC 793, September 1981, DOI 10.17487/RFC0793

[61] B. Tushir, H. Sehgal, R. Nair, B. Dezfouli, and Y. Liu, "The Impact of DoS Attacks onResource-constrained IoT Devices:A Study on the Mirai Attack", CoRR, April 2021

[62] A. Rao, R. Lanphier, and H. Schulzrinne, "Real Time Streaming Protocol (RTSP)", RFC 2326, April 1998, DOI 10.17487/RFC2326

[63] S. Dadkhah, H. Mahdikhani, P. K. Danso, A. Zohourian, K. A. Truong, and A. A. Ghorbani, "Towards the Development of a Realistic Multidimensional IoT Profiling Dataset", 2022 19th Annual International Conference on Privacy, Security &Trust (PST), Fredericton (Canada), August 22-24, 2022, pp. 1–11, DOI 10.1109/pst55820.2022.9851966

[64] M. U.Farooq, M. Waseem, S. Mazhar, A. Khairi, and T. Kamal, "A Review on Internet of Things (IoT)", International Journal of Computer Applications, vol. 113, March 2015, pp. 1–7, DOI 10.5120/19787-1571

[65] M. U.Farooq, M. Waseem, A. Khairi, and S. Mazhar, "A Critical Analysis on the Security Concerns of Internet of Things (IoT)", International Journal of Computer Applications, vol. 111, February 2015, pp. 1–6, DOI 10.5120/19547-1280

[66] Z. Huo, W. Zhu, and P. Pei, "Network Traffic Statistics Method for Resource-Constrained Industrial Project Group Scheduling under Big Data", Wireless Communications and Mobile Computing, vol. 2021, June 2021, pp. 1–9, DOI 10.1155/2021/5594663

[67] Information Sciences Institute of University of Southern California, "Internet Protocol", RFC 791, September 1981, DOI 10.17487/RFC0791

[68] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1", RFC 2616, June 1999, DOI 10.17487/RFC2616

[69] S. St.Laurent, M. Makoto, and D. Kohn, "XML Media Types", RFC 3023, January 2001, DOI 10.17487/RFC3023

[70] J. Postel, "Internet Control Message Protocol", RFC 777, April 1981, DOI 10.17487/RFC0777

[71] K. B. Egevang and P. Francis, "The IP Network Address Translator (NAT)", RFC 1631, May 1994, DOI 10.17487/RFC1631

[72] P. Eronen, Y. Nir, P. E. Hoffman, and C. Kaufman, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 5996, September 2010, DOI 10.17487/RFC5996

[73] D. N. Divyabharathi and N. G. Cholli, "A Review on Identity and Access Management Server (KeyCloak)", International Journal of Security and Privacy in Pervasive Computing, vol. 12, July 2020, pp. 46–53, DOI 10.4018/ijsppc.2020070104

[74] M. N. Aman, U. Javaid, and B. Sikdar, "Security Function Virtualization for IoT Applications in 6G Networks", IEEE Communications Standards Magazine, vol. 5, September 2021, pp. 90–95, DOI 10.1109/mcomstd.201.2100023

[75] M. Zolotukhin and T. Hamalainen, "On Artificial Intelligent Malware Tolerant Networking for IoT", 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Verona (Italy), November 27-29, 2018, pp. 1–6, DOI 10.1109/nfv-sdn.2018.8725767

[76] N. Guizani and A. Ghafoor, "A Network Function Virtualization System for Detecting Malware in Large IoT Based Networks", IEEE Journal on Selected Areas in Communications, vol. 38, June 2020, pp. 1218–1228, DOI 10.1109/jsac.2020.2986618

[77] M. Al-Shaboti, I. Welch, A. Chen, and M. A. Mahmood, "Towards Secure Smart Home IoT: Manufacturer and User Network Access Control Framework", 2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA), Krakow (Poland), May 16-18, 2018, pp. 892–899, DOI 10.1109/aina.2018.00131

[78] P. Massonet, L. Deru, A. Achour, S. Dupont, L.-M. Croisez, A. Levin, and M. Villari, "Security in Lightweight Network Function Virtualisation for Federated Cloud and IoT", 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud), Prague (Czech Republic), August 21-23, 2017, pp. 148–154, DOI 10.1109/ficloud.2017.43

[79] S. M. Kasongo, "An Advanced Intrusion Detection System for IIoT Based on GA and Tree Based Algorithms", IEEE Access, vol. 9, August 2021, pp. 113199–113212, DOI 10.1109/access.2021.3104113

[80] A. Basati and M. M. Faghih, "PDAE: Efficient network intrusion detection in IoT using parallel deep auto-encoders", Information Sciences, vol. 598, June 2022, pp. 57–74, DOI 10.1016/j.ins.2022.03.065

[81] R. K. Sharma and R. S. Pippal, "Malicious Attack and Intrusion Prevention in IoT Network using Blockchain based Security Analysis", 2020 12th International Conference on Computational Intelligence and Communication Networks (CICN), Bhimtal (India), September 25-26, 2020, pp. 380–385, DOI 10.1109/cicn49253.2020.9242610

[82] C. Jiang, J. Kuang, and S. Wang, "Home IoT Intrusion Prevention Strategy Based on Edge Computing", 2019 IEEE 2nd International Conference on Electronics and Communication Engineering (ICECE), Xi'an (China), December 9-11, 2019, pp. 94–98, DOI 10.1109/icece48499.2019.9058536

[83] D. G. V. Goncalves, F. L. de Caldas Filho, L. M. C. E. Martins, G. de O. Kfouri, B. V. Dutra, R. de O. Albuquerque, and R. T. de Sousa, "IPS architecture for IoT networks overlapped in SDN", 2019 Workshop on Communication Networks and Power Systems (WCNPS), Brasilia (Brazil), October 3-4, 2019, pp. 1–6, DOI 10.1109/wcnps.2019.8896297

[84] M. S. Haghighi, F. Farivar, and A. Jolfaei, "A Machine Learning-based Approach to Build Zero False-Positive IPSs for Industrial IoT and CPS with a Case Study on Power Grids Security", IEEE Transactions on Industry Applications, July 2020, pp. 1–9, DOI 10.1109/tia.2020.3011397

[85] J. R. Raj and S. Srinivasulu, "Design of IoT Based VPN Gateway for Home Network", 2022 International Conference on Electronics and Renewable Systems (ICEARS), Tuticorin (India), March 16-18, 2022, pp. 561–564, DOI 10.1109/icears53579.2022.9751838

[86] J. Fan, Z. Wang, and C. Li, "Design and Implementation of IoT Gateway Security System", 2019 International Conference on Artificial Intelligence and Advanced Manufacturing (AIAM), Dublin (Ireland), October 16-18, 2019, pp. 156–162, DOI 10.1109/aiam48774.2019.00039

[87] S. Alharbi, P. Rodriguez, R. Maharaja, P. Iyer, N. Subaschandrabose, and Z. Ye, "Secure the internet of things with challenge response authentication in fog computing", 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC), San Diego (California, USA), December 10-12, 2017, pp. 1–2, DOI 10.1109/pccc.2017.8280489

[88] C. Zedak, A. Lekbich, A. Belfqih, J. Boukherouaa, T. Haidi, and F. E. Mariami, "A proposed secure remote data acquisition architecture of photovoltaic systems based on the Internet of Things", 2018 6th International Conference on Multimedia Computing and Systems (ICMCS), Rabat (Morocco), May 10-12, 2018, pp. 1–5, DOI 10.1109/icmcs.2018.8525902

[89] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks", American Institute of Chemical Engineers Journal, vol. 37, February 1991, pp. 233–243, DOI 10.1002/aic.690370209