

# UNIVERSITY OF SALERNO

DEPARTMENT OF COMPUTER SCIENCE



Master's Degree in Computer Science

Curriculum in Cybersecurity

## Evaluation of a MCU-Deployed Siamese Neural Network-based IDS using a real IoT attack dataset

Rapporteur:

**Francesco PALMIERI**

Candidate:

**Alberto MONTEFUSCO**

**Mat. 0522501498**

Correlator:

**Massimo FICCO**

ACADEMIC YEAR 2023/2024

*Memory is the intelligence  
of the dumb.*

# Contents

<b>Introduction</b>	<b>3</b>
Problem Statement . . . . .	3
Motivation and Objectives . . . . .	3
Structure of the Thesis . . . . .	3
<b>1 State of the Art</b>	<b>4</b>
1.1 The Internet of Things . . . . .	4
1.2 Types of IoT Architectures . . . . .	5
1.2.1 Hardware IoT Architectures . . . . .	5
1.2.2 Software IoT Architecture . . . . .	6
1.2.3 General IoT Architectures . . . . .	10
1.3 Security in IoT . . . . .	11
1.3.1 Security Challenges in the IoT Domain . . . . .	12
1.3.2 Taxonomy of Security Attacks . . . . .	13
1.3.3 Defense and Prevention Methodologies . . . . .	14
1.4 Intrusion Detection Systems . . . . .	16
1.4.1 Types of IDS . . . . .	17
1.4.2 Machine Learning Techniques vs Deep Learning Approaches . . . . .	18
1.4.3 Overview of Existing Datasets . . . . .	20
1.4.4 Challenges in Implementing IDS on Embedded Systems . . . . .	25
<b>2 My Cloud-Based IoT Architecture</b>	<b>27</b>
2.1 Architecture Design . . . . .	27
2.1.1 Hardware components . . . . .	28
2.2 Network Creation and Device Communication . . . . .	33
2.2.1 Initial configurations . . . . .	33
2.2.2 Insecure Network . . . . .	39
2.2.3 Secure Network . . . . .	42
2.3 Configuration of IoT Devices . . . . .	45

<b>3</b>	<b>Creation of a Real IoT Attack Dataset</b>	<b>47</b>
3.1	Types of Attacks Performed . . . . .	47
3.1.1	Denial of Service Attacks . . . . .	48
3.1.2	Brute Force Attack . . . . .	49
3.1.3	MQTT Protocol Attacks . . . . .	49
3.1.4	Reconnaissance Attacks . . . . .	51
3.2	Logging Network Traffic . . . . .	54
3.2.1	Architecture for Traffic Sniffing . . . . .	54
3.2.2	The Dataset in detailed . . . . .	55
3.2.2.1	Scan times . . . . .	55
3.2.2.2	Splitting the dataset . . . . .	56
3.2.2.3	Features Selection . . . . .	56
3.2.2.4	PCAP to CSV . . . . .	58
3.3	Comparison between TON_IoT and my Dataset . . . . .	59
3.3.1	Feature Selection and Data Pre-processing . . . . .	59
3.3.2	Number of selected Samples . . . . .	59
3.3.3	Summary of Datasets Differences . . . . .	61
<b>4</b>	<b>Construction of an Intrusion Detection System</b>	<b>62</b>
4.1	Data Pre-processing . . . . .	63
4.1.1	Data Cleaning . . . . .	63
4.1.2	Data Transformation . . . . .	65
4.1.3	Data Normalization . . . . .	65
4.2	Experimental Setup . . . . .	66
4.2.1	Machine Learning Models . . . . .	66
4.2.1.1	Support Vector Machine (SVM) . . . . .	67
4.2.1.2	Random Forest (RF) . . . . .	67
4.2.1.3	K-Nearest Neighbors (KNN) . . . . .	67
4.2.2	Siamese Neural Network . . . . .	70
4.2.2.1	Generation of Pairs . . . . .	70
4.2.2.2	Neural Network Architecture . . . . .	74
4.2.2.3	Concept of Similarity and Distance . . . . .	76
4.2.2.4	Train of Siamese Network . . . . .	77
4.2.2.4	Motivation of the Siamese Neural Network in the Thesis . .	78
4.2.3	Transfer Learning Approach . . . . .	78
<b>5</b>	<b>Evaluation of Results</b>	<b>80</b>
5.1	Performance of Machine Learning Models . . . . .	80
5.1.1	Train and Test Classifiers . . . . .	81
5.1.2	Evaluating pre-trained Classifiers . . . . .	84

---

## CONTENTS

---

5.1.3	Report of results obtained . . . . .	87
5.2	Performance of Siamese Networks . . . . .	89
5.2.1	Results of my Dataset with 4 Classes . . . . .	89
5.2.2	Results of my Dataset with 5 Classes . . . . .	90
5.2.3	Results of TON_IoT Dataset . . . . .	91
5.2.4	Evaluating pre-trained Siamese Network . . . . .	92
5.2.5	Report of results obtained . . . . .	93
5.3	Performance of Transfer Learning . . . . .	94
5.3.1	Pre-Train on 4-Class Dataset and Fine-Tuning on TON_IoT . . . .	94
5.3.2	Pre-Train on 5-Class Dataset and Fine-Tuning on TON_IoT . . . .	95
5.3.3	Report of results obtained . . . . .	95
5.4	Comparative Analysis . . . . .	96
<b>6</b>	<b>MCU-Deployed SNN for IDS</b>	<b>98</b>
6.1	Embedded Hardware Selection . . . . .	99
6.2	Model conversion for Embedded Environments . . . . .	99
6.3	Testing and Validation on Embedded Hardware . . . . .	101
6.4	Results . . . . .	104
6.5	Problems and Solutions . . . . .	105
<b>7</b>	<b>Conclusions and Future Developments</b>	<b>108</b>
7.1	Summary of Obtained Results . . . . .	108
7.2	Study Limitations . . . . .	109
7.3	Proposals for Future Research . . . . .	109
	<b>Bibliography</b>	<b>112</b>
	<b>List of Figures</b>	<b>116</b>

---

# Introduction

## Problem Statement

The rapid expansion of the Internet Of Things (IoT) has definitely brought a revolution in data collection, transmission and analysis in various industries, including but not limited to healthcare, smart cities and industrial automation. For example, IoT devices such as wearable health monitors track continuous data regarding a patient's health in healthcare; similarly, sensors embedded within cities help optimize energy use and manage traffic flow in smart cities. IoT is applied in industrial automation for predictive maintenance to optimize production processes. Paralleling this technological growth, remarkable vulnerabilities were also introduced into the architectures in IoT, most due to the little consideration of security issues in most devices online. Such devices normally have a limited capability of computation, low memorization and low power, hence presenting difficulties in the implementation of advanced mechanisms of security [1].

Thus, this rendered devices vulnerable to various cyberattacks, such as DDoS attacks, MITM attacks and malware infiltration. For example, the 2016 Mirai botnet attack took advantage of insecure IoT devices in launching a massive DDoS attack that caused widespread outages in major internet services [2]. Such vulnerabilities not only put at risk the integrity and confidentiality of data sent but also jeopardize the overall functionality and reliability of critical systems needing IoT infrastructures. Furthermore, this very heterogeneous nature of IoT ecosystems, inherently containing a vast array of devices from various different manufacturers with disparate standards for security, threatens to exacerbate this challenge of security completeness. It is easier for bad actors to proliferate points of entry that can be exploited due to the lack of standardized safety frameworks and protocols among IoT devices.

This fragmentation makes it very difficult to deploy integrated security solutions, increasing the risk of cascading coordinated attacks on interdependent systems. With critical infrastructures such as health systems, transportation networks and industrial control systems depending on IoT technologies, it is equally important to provide security to these networks. Apart from information breaches, a breach in IoT security has a number of other potential consequences, including physical damage, loss of life and huge economic losses. For example, successful cyber-attacks on IoT-based medical devices could mean

wrong dosages administered to patients, while those on smart grid systems result in disrupted electricity supply to entire regions. This, therefore, calls for defending IoT networks against such pervasive threats with considerable importance and dedicated joint effort. Most current security solutions inadequately address the challenges of IoT environments, especially in scalability, real-time threat detection and resource efficiency. Advanced IDS fits to the constraints and requirements of IoT architectures are urgently needed. Such systems should be able to detect as well as mitigate established and emerging threats with a limited resource of devices through the introduction of minimal computational overhead. Conclusively, while IoT technologies promise transformative benefits in many spheres, the burgeoning security vulnerabilities accompanying these expose considerable risks that ought to, where possible, be systematically brought under control. Effective security mechanisms like AI-driven Intrusion Detection Systems form a proactive approach toward ensuring the integrity, availability and confidentiality of IoT networks, hence reliable operation within critical infrastructures.

## Motivation and Objectives

The motivation for this thesis is the growing demand for cybersecurity solutions in the expanding world of IoT devices. Although IoT architectures seem predominant today, they provide new vulnerabilities that can be used by both known and emerging cyber threats. For this reason, a new dataset of attacks has been created that can be used by the literature to build artificial intelligence systems to detect and react to attacks against IoT devices. Therefore, the main objective of this thesis is to develop an up-to-date attack dataset that fits the context of IoT systems to facilitate more effective AI-based detection and mitigation strategies.

Another important objective of this thesis is to design and test ML classifiers and a Siamese neural network as an IDS, capable of recognizing most known attacks in IoT networks. Both the Siamese network and the ML classifiers were tested in both environments, with and without the MQTT traffic generated by the IoT devices, to understand its adaptability and robustness. Furthermore, they were also trained using the TON\_IoT dataset to demonstrate that the dataset created offers superior performance compared to an already established dataset in the literature. This comparison highlights the validity of the created dataset and its potential use in academic and research contexts, as it provides equally reliable results for classification and anomaly detection in IoT systems. Finally, the trained Siamese network model was implemented on an ESP32 embedded system to verify its practical effectiveness. This involved subjecting it to repeated attacks, testing its response time for effective detection and response to attacks in real-world scenarios, with the aim of obtaining a lightweight yet reliable IDS suitable for IoT.

## Structure of the Thesis

In this section, I will explain how the thesis was structured, in particular, I will briefly explain what is covered in each chapter:

1. **"State of the Art"**: the first chapter reviews existing literature on IoT security, including architectures, common threats and vulnerabilities, machine learning techniques used for Intrusion Detection System, the datasets used for these AI models and implementations of neural networks on embedded systems.
2. **"My Cloud-Based IoT Architecture"**: in the second chapter, the implementation of a Cloud-Based IoT architecture was discussed, with a special focus on network construction. In this section, both a secure architecture, with security measures implemented, and an insecure one were presented, highlighting the differences in terms of vulnerability and protection. This comparison is critical to understanding the importance of security in IoT networks and the potential consequences of inadequate management.
3. **"Creation of a Real IoT Attack Dataset"**: the third chapter describes which attacks were carried out, the methods and tools used to carry out these real attacks on the implemented IoT architecture and how all the generated traffic flow was captured and saved to create the dataset. Furthermore, a technical and structural comparison was made between the realized dataset and TON\_IoT.
4. **"Construction of an Intrusion Detection System"**: the fourth chapter describes the implementation of an IDS through the development of ML classifiers and a Siamese Neural Network. Both were also trained with the TON\_IoT dataset in order to compare them with the dataset created. This comparison allows us to evaluate the performance of my dataset against an established standard, demonstrating its effectiveness and validity for academic research and IoT applications.
5. **"Evaluation of Results"**: the fifth chapter presents an in-depth analysis of the performance of Machine Learning models, Siamese Neural Networks and the application of Transfer Learning.
6. **"MCU-Deployed SNN for IDS"**: the sixth chapter deals with the conversion of the Siamese neural network into a compatible format for the *ESP32* embedded device; the entire procedure is explained followed by various tests, results and problems encountered.
7. **"Conclusions and Future Developments"**: the thesis concludes with chapter seventh, summarizing the contributions of this work and suggesting directions for future research aimed at improving IoT security.



# Chapter 1

## State of the Art

In this chapter, I will discuss the state of the art related to the Internet of Things (IoT). First, the main types of IoT architectures will be presented, with a focus on cloud-based solutions, which represent one of the most popular implementations due to their ability to efficiently manage large amounts of data and devices. Next, the chapter will also focus on security aspects in the IoT, including the most common threats and vulnerabilities inherent in IoT systems. In particular, defence and prevention methodologies will be introduced, with a focus on Intrusion Detection Systems (IDS), which are a critical line of defence for protecting IoT environments. Approaches based on machine learning and deep learning techniques applied to IDSs will be compared, and the main existing datasets will be analyzed. Finally, the challenges of implementing IDSs on embedded devices will be discussed, considering the resource constraints and integration difficulties in these contexts and their implementation within embedded systems.

### 1.1 The Internet of Things

The Internet of Things (IoT) is formally defined as:

*"a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies." [3]*

From this definition, IoT is considered a worldwide network that supports the collection, transmission and processing of information, contributing to the development of the digital society. It enables advanced and innovative services that improve efficiency, productivity and quality of life in various sectors such as healthcare, energy, transport and manufacturing. The IoT connects physical devices such as sensors, home appliances and industrial machinery, as well as virtual entities such as software applications and online services. It also utilizes both current and developing ICT technologies, ensuring that different systems and devices can communicate and work together seamlessly through interoperability.

### 1.2 Types of IoT Architectures

These types of IoT devices produce trillions of data every day, but have little value if this data cannot be efficiently collected, stored, analysed and communicated. This is why having an effective IoT architecture to collect, store and analyse IoT data is crucial. In the literature, there exist a lot of works proposing new IoT architectures applied on a specific or many application domains. IoT architecture may consist physical objects (e.g. sensors, actuators), virtual objects (e.g. cloud services, communication layers and protocols) or a hybrid of these two perspectives. These architectures must be able to support IoT devices and services, as well as the workflow that these devices will affect. Thereby, IoT architectures are classified as hardware architectures, software architectures and general architectures.

#### 1.2.1 Hardware IoT Architectures

Many hardware architectures have been proposed to support the distributed computing environment required by the Internet of Things. Among these architectures we find the peer-to-peer architecture, the EPC based architecture and the Wireless Sensor Networks (WSNs) based architecture. In the following, I discuss those three hardware architectures.

##### **Peer to Peer Architecture**

In this model, the interaction between processes/peers/devices is symmetric: each process will act as a client and a server at the same time (acting as a 'servant'). Peer-to-peer architectures can be built using a distribution protocol such as the multiple Distributed Hash Table (DHT) routing protocol. It is possible to design a P2P architecture to be especially beneficial for Web of Things (WoT) applications, like M2M (Machine to Machine) communication, involving embedded devices.

##### **EPC Based Architecture**

EPC (Electronic Product Code) is an universal identifier that gives a unique identity to an item a RFID tag is affixed to. The identity is made to be unique so that each object is identifiable within the objects field [4]. The EPC number enables data information exchange among companies and their business partners. For standardization purposes, an architecture known as the EPC global network was proposed. In this architecture, roles, interfaces and a common vocabulary are specified, leaving implementation details for end users depending on the application domain. An EPC based architecture could be built over an heterogeneous access network, particularly using a ZigBee network as it can collect the latest information about 'Things' [5]. The proposed architecture provides two functions:

- the first one is how to register new objects or devices to a home area network;
- the second one is how to make objects communicate through the Internet with generic protocols.

Among the difficulties encountered using this architecture are the high cost of purchasing hardware (such as RFID tags and readers), the ability to track unique objects can raise privacy issues, especially if products are associated with specific individuals without adequate data protection measures. The large amount of data generated by EPC-based IoT devices can make information management, storage and analysis complex. RFID tags may have reading problems due to interference, signal-blocking materials or adverse environmental conditions, affecting the reliability of the system. Furthermore, although EPC is standardised, incompatibilities with other identification systems or local standards may arise, complicating integration with existing infrastructures.

### Sensors and WSNs Based Architecture

Wireless sensor networks (WSN) allow embedded devices to be connected and used in a seamless way. Using WSN while designing IoT architectures is very promising since it helps implementing distributiveness and context-awareness which are main features of IoT architectures. While designing an IoT architecture based on WSNs, it is possible to include a complete IP adaptation method, as it is the case for the Sensor Networks for an All-IP World (SNAIL) architecture which includes four significant network protocols: mobility, web enablement, time synchronization, and security. Another existing approach relies on using M2M gateway in the IoT architecture based on WSNs. The main idea is to connect different sensors to the M2M gateway for communication with end users or different provided services. This solution can be applied in smart building applications using WSNs. Heterogeneity and security issues are fulfilled using this approach. WSNs are also widely used in smart cities in order to manage smart traffics and mobility.

The disadvantages of WSN architectures relate to energy limitations, as battery-powered nodes have a limited lifespan and low bandwidth with reduced transmission rates. In addition, they are vulnerable to security due to limited protection mechanisms and exhibit scalability and reliability difficulties due to the complexity of managing many nodes and environmental interference.

### 1.2.2 Software IoT Architecture

Software architectures are necessary to ensure access and sharing of services offered by IoT devices. There are several approaches to provide application framework for IoT such as SOA, RESTful and architectures based on fog and cloud computing. These architectures focus on services and flexibility and cover Operating systems, IoT middleware, APIs, Data

management, Big data, etc. In the following, I discuss SOA based architecture, RESTful architecture and cloud/fog based architecture.

### SOA Based IoT Architecture

Service Oriented Architecture (SOA) is a software architectural style that is commonly built using web services standards. It is also possible to implement SOA using any other service-based technology, such as Jini, CORBA or REST. In SOA based IoT architecture, each device is a service consumer and/or a service provider offering services or sharing resources and interacting with service consumers via compatible service APIs (Application Programming Interfaces). SOA technologies enable publishing, discovery, selection, and composition of services offered by IoT devices. Unlike traditional enterprise services and applications, which are mainly virtual entities, real-world services are provided by embedded systems that are related directly to the physical world. In IoT architectures, we can find both types of services according to the application domain (Figure 1.1).

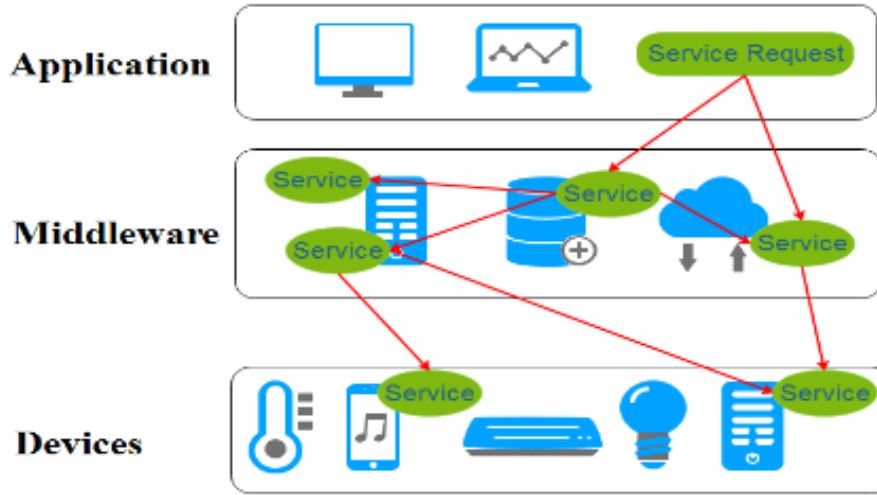


Figure 1.1: SOA based IoT architecture.

The disadvantages of this architecture include implementation complexity, performance and latency issues, device resource limitations, security challenges, scalability difficulties and high development and maintenance costs.

### REST Based IoT Architecture

The Representational State Transfer (REST) is a software architectural style that defines a set of constraints to be used for creating web services. REST architecture is mainly based on constrained client-server communication. REST is implemented by Universal Resource Indicators (URIs) for identifying and Extensible Markup Language (XML) to exchange data. The RESTful architecture is known to be loose-coupled, simple and scalable, which motivate to use Web standards to interact with smart things. As a result, the concept of Web of Things (WoT) is introduced rather than Internet of Things (IoT).

In the Web of Things concept, smart things and their services are fully integrated in the Web by reusing and adapting technologies and patterns commonly used for traditional Web content. It should be noted that HTTP introduces a communication overhead and increases average latency. It should be used for pervasive scenarios where relatively longer delays do not affect the system requirements.

It is possible to build web services for IoT applications by using the Constrained Application Protocol (CoAP): it allows REST based communications among applications residing in distributed and networked embedded systems. The CoAP aims to provide a protocol stack able to cope with limited packet sizes, low energy devices and unreliable channels, which are the main characteristics of IoT architectures.

Another existing protocol used to implement the REST architecture is the Message Queue Telemetry Transport (MQTT). MQTT is a lightweight event- and message-oriented protocol, which allows the devices to asynchronously communicate across constrained networks to reach remote systems. MQTT is based on a publish/subscribe interaction pattern. In particular, MQTT has been implemented for easily connecting the things to the web and support unreliable networks with small bandwidth and high latency. As a result, MQTT is adequate for designing REST based IoT architectures. The main issue with the MQTT protocol is low level security. There exist some work to enhance the security of the MQTT protocol by proposing an Open Source AUthenticated Publish/Subscribe (AUPS) system for the Internet of Things.

### **Cloud-Based IoT Architectures**

IoT systems generate a huge amount of data that has to be stored, processed and presented in a seamless, efficient and easily interpretable way. Cloud computing provide high reliability, scalability, and autonomy to IoT systems. In fact, a cloud based platform acts as a receiver of data from the ubiquitous sensors, as a computer to analyze and interpret data, as well as a visualizations web based tool. The National Institute of Standards and Technology (NIST) defines cloud computing as a model that provides on-demand access to shared computing resources, including networks, servers, storage, and applications. It is possible to design an IoT architecture based on a Cloud centric vision. According to this vision, a conceptual framework integrating the ubiquitous sensing devices and the applications is proposed (Figure 1.2).

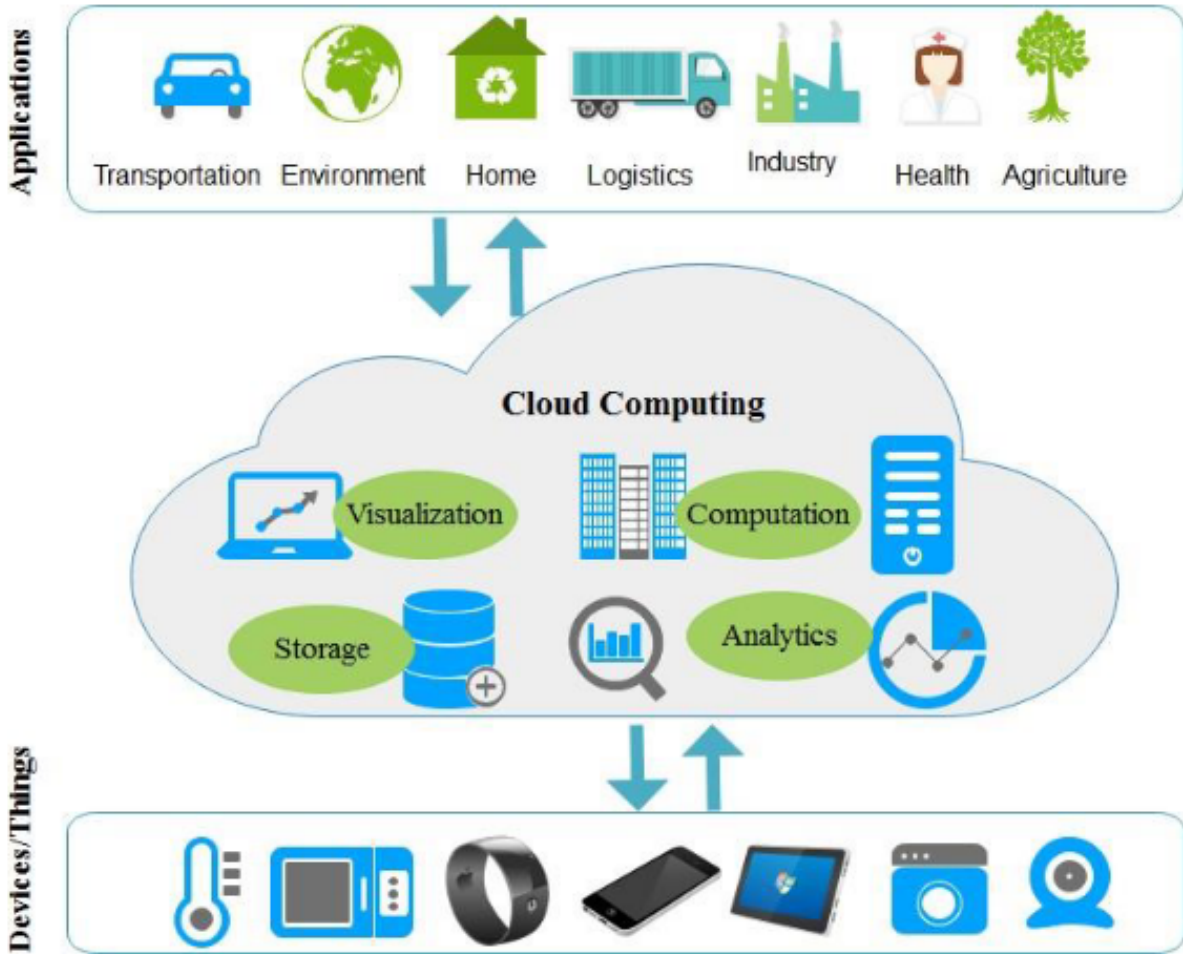


Figure 1.2: Cloud based IoT architecture.

There are four main deployment models for cloud infrastructure:

1. *Private Cloud*: used exclusively by one organization, offering security and control but at a higher cost.
2. *Public Cloud*: accessible by multiple users via the internet, though it carries security risks.
3. *Hybrid Cloud*: combines multiple cloud models to maximize benefits.
4. *Community Cloud*: shared by organizations with common interests or requirements, hosted internally or externally.

Cloud computing services can be divided into three main types:

1. *Infrastructure as a Service (IaaS)*: provides virtualized computing resources.
2. *Platform as a Service (PaaS)*: offers a platform allowing customers to develop, run, and manage applications.
3. *Software as a Service (SaaS)*: provides software applications over the internet.

Among the disadvantages, we have that the cloud-centric view is criticized for being very centralized. Indeed, physical devices must be able to communicate with cloud services that are typically geographically remote and scattered. Such an aspect can be very restrictive since the devices used in IoT systems generally have very limited resources. To facilitate access to cloud services, there are architectures that provide the use of access points as intermediates between physical devices and cloud services. This is called the fog computing. The Figure 1.3 illustrates an architecture based on the fog computing.

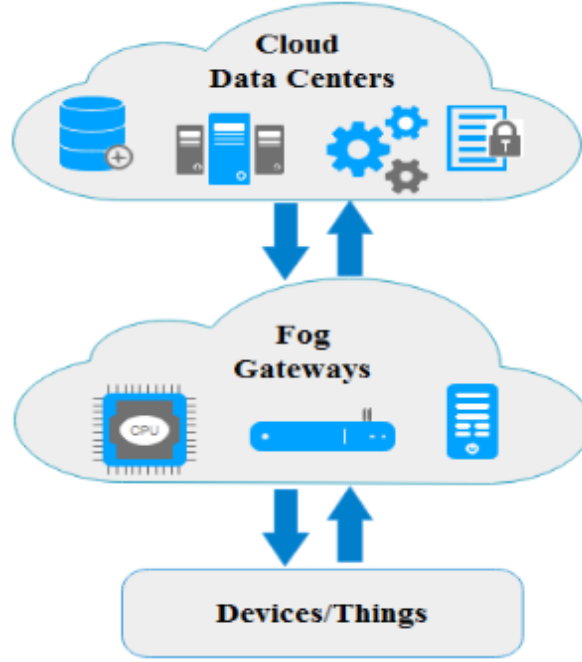


Figure 1.3: Cloud/Fog based IoT architecture.

### 1.2.3 General IoT Architectures

There is no single, standardized architecture for the Internet of Things (IoT); several layered architectures have been proposed to offer end-to-end solutions tailored to the specific requirements of various use cases. Examples include the DIAT architecture, which addresses challenges such as scalability and interoperability across three main layers (VOL, CVOL and SL) and the five-layer architecture for the Internet of Vehicles (IoV), which integrates artificial intelligence and specific protocols to ensure security and connectivity. Other proposals present generic conceptual architectures applicable to numerous sectors such as smart agriculture, logistics, healthcare and smart home, often without detailed practical implementations. Some research offers modular architectures that can be customized and combined without affecting overall functionality, providing technology choices and experimental results in different application domains [6].

### 1.3 Security in IoT

The number and variety of IoT devices have rapidly grown in the last years, infact, thanks to a plethora of new 'smart' services and products, such as smart appliances, smart houses, smart watches, smart TVs, and so on, the IoT devices are quickly spreading in all environments, becoming everyday more pervasive. Moreover, many of such smart services require users to intentionally reveal some personal (and, sometimes, private) information in change for advanced and more personalized services. It is then clear that security and privacy should be of primary importance in the design of IoT technologies and services. Unfortunately, this is not the case for many IoT commercial products that are provided with inadequate, incomplete, or ill-designed security mechanisms.

In the last years, growing attention has been dedicated to the risks related to the use of simple IoT devices in services that have access to sensitive information or critical controls, such as, video recoding of private environments, real-time personal localization, health-monitoring, building accesses control, industrial processes and traffic lights. Furthermore, some security attacks against commercial IoT devices have appeared in the mass media, contributing to raise public awareness of the security threats associated with the IoT world.

In order to make commercial IoT devices more resilient to cyber attacks, security should be taken into account right from the design stage of new products. However, the wide heterogeneity of IoT devices hinders the development of well established security-by-design methods for the IoT. The challenge is further complicated by the severe limits in terms of energy, communication, computation, and storage capabilities of many IoT devices. Such limits indeed prevent the possibility of adopting standard security mechanisms used in more traditional Internet-connected devices and call for new solutions that, however, are not yet standardized. Besides the technical aspects, it is also necessary to develop a cyber-security culture among the IoT stakeholders, in particular manufacturers and final users. As a matter of fact, many IoT device manufacturers come from the market of low-cost sensors and actuators (e.g., home automation, lights control and so on). Such devices were originally designed to work in isolated systems, for which the security threats are much more limited. As a consequence, many manufacturers do not possess a solid expertise in cybersecurity and may be unaware of the security risks associated with connecting their devices to a global network. Such a lack of know-how, together with the hectic approach to the design of new products and the need to compress costs and time-to-market have led to the commercialization of IoT products where security is either neglected or treated as an afterthought. In parallel, the final users are also not much educated in terms of security practices and often fail to implement even the most basic procedures to protect their devices as, e.g., changing the preinstalled password of the devices on first use. To this end, I will discuss the origins of some threats and the possible counteractions.



### 1.3.1 Security Challenges in the IoT Domain

The attacks against IoT devices are often simple and easy to conduct. They could be performed in order to break user privacy and leak personal sensible information. The collected data can indeed range from simple room temperature and humidity measurements, to more sensible information such as the heart-rate signal, or the user's location and living habits. Another common attack strategy consists in compromising one device in the IoT network and use it as a beachhead to perform fraudulent acts toward another network node.

To begin with, I present a taxonomy of the security requirements for an IoT system with respect to the different operational levels, that is to say, at the Information, Access and Functional level.

#### Information Level

At this level, security should guarantee the following requirements.

1. *Integrity*: the received data should not been altered during the transmission.
2. *Anonymity*: the identity of the data source should remain hidden to third parties.
3. *Confidentiality*: data cannot be read by third parties. A trustworthy relationship should be established between IoT devices in order to exchange protected information. Replicated messages must also be recognizable.
4. *Privacy*: the client's private information should not be disclosed during the data exchange. It must be hard to infer identifiable information by eavesdroppers.

#### Access Level

It specifies some security mechanisms to control the access to the network. More specifically, it provides the following functionalities.

1. *Access Control*: it guarantees that only legitimate users can access to the devices and the network for administrative tasks (e.g., remote reprogramming or control of the IoT devices and network).
2. *Authentication*: it checks whether a device has the right to access a network and whether a network has the right to connect the device. This is likely the first operation carried out by a node when it joins a new network. Note that devices have to provide strong authentication procedures in order to avoid security threats.
3. *Authorization*: it ensures that only the authorized devices and the users get access to the network services or resources.

### Functional Level

This level defines the security requirements in terms of the following criteria.

1. *Resilience*: it refers to network capacity to ensure security for its devices, even in case of attacks and failures.
2. *Self Organization*: it denotes the capability of an IoT system to adjust itself in order to remain operational even in case of failure of some parts due to occasional malfunctioning or malicious attacks.

### 1.3.2 Taxonomy of Security Attacks

Besides the requirements and mechanisms at the information, access, and functional levels, it is important to understand which are the vulnerabilities and the possible attacks at the different layers of the communication stack. As explained in section 1.2, the communication architecture of an IoT system can be roughly divided in Edge, Access, and Application layers. The edge layer provides PHY and MAC functionalities for local communications. The access layer grants the connection to the rest of the world, usually through a gateway device and a Middleware Layer that acts as intermediary between the IoT world and the standard Internet. Finally, the Application Layer takes care of the service-level data communications. In the following I present a possible taxonomy of the attacks that can target these communication layers.

#### Edge Layer

One of the main threats at this level is represented by the *side channel attacks*. The goal of these attacks is to leak information from the analysis of side signals, such as power consumption, electromagnetic emissions, and communication timing, while nodes are performing encryption procedures. Among them, the power consumption of the devices is widely exploited to guess and recover the encryption secret keys. At the edge layer, IoT devices are also vulnerable to *hardware trojan* and *DoS attacks* that attempt to make resources unavailable to the legitimate users, e.g., by forcing the device to exit sleep (low-power consumption) mode in order to drain their batteries, or by jamming the radio communications. Also, the device package can be tampered with, e.g., to extract the cryptographic secrets of the device, modify its software to disguise a malicious node as a legacy one (camouflage), or attempt reverse engineering to figure out the details of proprietary communication protocols and possibly reserved information (as patent-covered algorithms).

### Access/Middleware Layer

At this level the main attacks are *eavesdropping* (also called sniffing), injection of fraudulent packets and nonauthorized conversations. Even *routing attacks* have to be taken into account: an attacker may use this kind of attack to spoof, redirect, misdirect, or drop data packets.

### Application Layer

Attacks at the Application Layer are quite different from the previous ones, since they directly target the software running on the devices rather than the communication technology. Such attacks may address the integrity of, e.g., machine learning algorithms, where the attacker manipulates the training process of the learning algorithm to induce misbehaviors. There can also be attacks on the login and authentication phases.

### 1.3.3 Defense and Prevention Methodologies

In this paragraph, I present standard security mechanisms that have been designed to satisfy the requirements described in the previous section 1.3.1.

#### Encryption

It is the main and most important operation to ensure confidentiality during the communication. It consists in changing the actual message (plaintext) into a different one (ciphertext) using a hash function that can be easily reverted only knowing a secret key. Using encryption, a possible eavesdropper can only have access to the ciphertext, but should not be able to interpret the content of the message. The encryption mechanism can be symmetric or asymmetric. In symmetric encryption, the same secret key is used both for message encryption and decryption, and hence it must be known by both the sender and the receiver. In the asymmetric case, each endpoint needs to possess its own pair of keys: a public key and the associated private key, which cannot be easily derived from the public one. The public key can be known to anyone, while the private key should be kept secret. The public and private keys are designed in a way that a message encrypted with the former can only be decrypted with the latter. Therefore, to guarantee confidentiality, the message is encrypted by the sender by using the public key of the receiver, which can then recover the original message by using its own private key.

#### Lightweight Cryptography

Given the growth of the number of connected, low-complexity IoT devices, the research community has tried to design specific security algorithms for resource and energy constrained devices. Lightweight cryptography is a new branch of cryptography that focuses

on these aspects, including new encryption block and stream ciphers, message authentication codes, and hash functions, which are conceived to be executed by devices with limited computation, communication, and storage capabilities. In 2012 the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) published the ISO/IEC 29192 standard that specifies a series of lightweight encryption mechanisms, included the block ciphers PRESENT and CLEIFA. PRINCE is another lightweight block cipher, not included in the standard [7]. As lightweight hash function, ISO/IEC 29192 standard proposed PHOTON [8] and SPONGENT [9]. In 2013 NIST started a lightweight cryptography project to investigate and develop solutions for real-world applications. At the beginning of 2019 NIST has published a call for algorithms for lightweight cryptography: after discussion and evaluation, the algorithms will go through a standardization process [10].

### Random Number Generators

An important aspect for security is the randomness: security protocols frequently require the generation of (pseudo)random numbers for different purposes as, e.g., to create nonces during the authentication phase, to avoid replay attacks, and to generate asymmetric keys. A random number generator is cryptographically secure when it produces a sequence for which no algorithm can predict in polynomial time the next bit of the sequence from the previous bits, with a probability significantly greater than  $(1/2)$ . According to Shannon's mathematical theory of communication, the entropy of a  $k$ -bit long (pseudo)random sequence must be as close as possible to  $k$ . Two types of random number generators are commonly used for cryptographic applications:

1. the true random number generator (TRNG) that exploits physical noise sources;
2. the pseudo random number generator (PRNG) that expands a relatively short key into a long sequence of seemingly random bits, using a deterministic algorithm.

PRNGs are typically used in real applications and technologies. In this case, since the adopted algorithms are usually known, the seed of the pseudorandom generator is the only source of randomness and, as such, it must be properly selected. Unfortunately, most of the source of randomness available in laptops and desktop PCs are not available in low-end embedded systems.

### Secure Hardware

As discussed in the previously, IoT devices are vulnerable to edge layer attacks, in particular to side channel attacks. *Physically Unclonable Functions* (PUFs) can be adopted to improve hardware security. The basic concept of PUF is to exploit little differences introduced by the fabrication process of the chip to generate a unique signature of each

device. A PUF circuit provides a response to a given input challenge and, due to the intrinsic hardware differences, the responses are chip specific.

PUFs can be categorized into strong and weak [11]. If a PUF can support a number of challenge-response pairs that are exponential in the number of challenge bits, it is called strong PUF. Strong PUFs are typically used for authentication protocols that require new pairs for each operation. On the other hand, weak PUFs can support a small number of challenge-response pairs and they are used for cryptographic key generation, avoiding the need to store secure keys on the devices [12].

### Intrusion Detection Systems

As discussed above, different security mechanisms have been proposed to protect the devices against threats at the different layers. However, besides preventing the attacks, it is also fundamental to be able to detect ongoing attacks. Complex antivirus software and traffic analyzers cannot be used in IoT devices, due to resource and energy constraints. For this, lightweight intrusion detection methods have been presented in the last years. For example, anomalies in system parameters, like CPU usage, memory consumption, and network throughput, may be indicative of an ongoing attack. Machine learning can also be exploited for intrusion detection purposes. For example, a random forest classification algorithm is used to group the traffic flows into different categories, based on some selected features. An attack is detected when some flows exhibit nonstandard patterns and are hence classified as anomalous [13].

## 1.4 Intrusion Detection Systems

In recent years, academic research has focused on developing innovative, effective and efficient intrusion detection systems for the IoT. In particular, the application of Machine Learning (ML) and Deep Learning (DL) techniques has opened up new perspectives for improving the security of IoT devices. Numerous studies have explored the use of ML and DL-based IDS, highlighting how these techniques can increase the ability to detect sophisticated and previously unknown attacks. However, practical implementation of such solutions presents several challenges, including the need for adequate datasets for training and model evaluation, as well as hardware restrictions of embedded systems.

To fully understand the potential and limitations of ML and DL-based IDS in the IoT, it is crucial to analyze the differences between these techniques and how they affect the performance of sensing systems. In addition, the availability and quality of datasets used in the literature play a crucial role in the effectiveness of learning algorithms. Finally, addressing the challenges in implementing IDS on embedded systems is essential to ensure viable solutions that can operate effectively within the resource constraints typical of IoT devices.

### 1.4.1 Types of IDS

Intrusion detection system (IDS) comes in many shapes and sizes, where some are simply software applications that run on servers or workstations. Their main purpose is to monitor events on systems or networks and notify the security administrators of any events that is determined to be worthy of alert by the sensors. There are several types of IDS that can be used to aid the security administrators as shown in Figure 1.4.

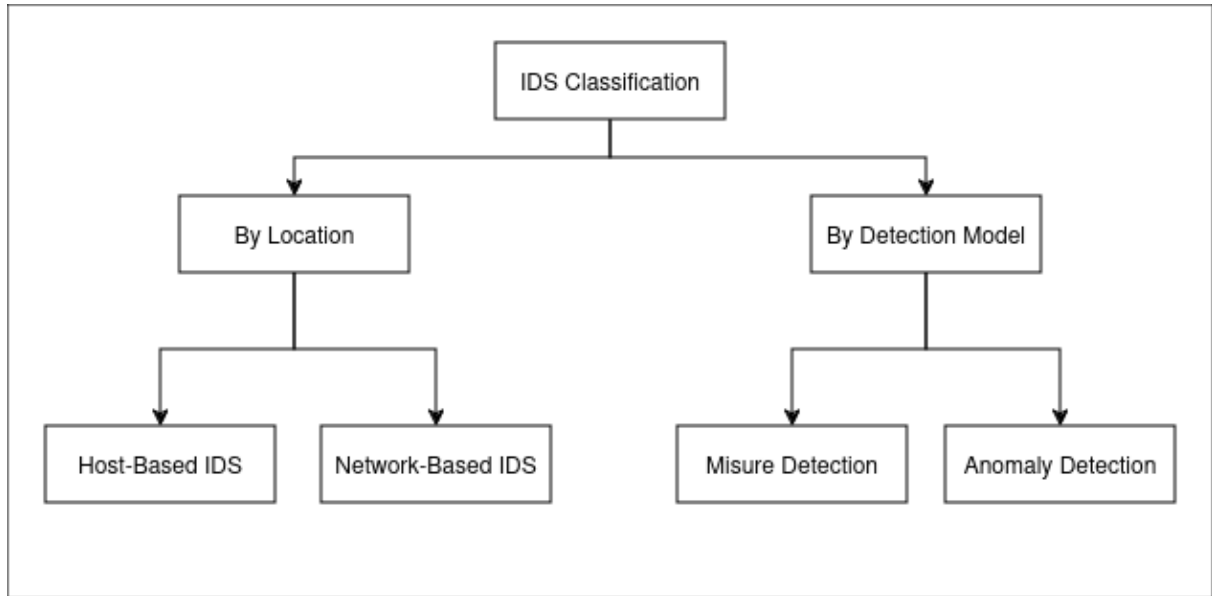


Figure 1.4: IDS Classification.

Network-based intrusion detection systems (NIDS) are devices that are distributed within networks to monitor the traffic to detect abnormal activities, such as attacks against hosts or servers. At first, NIDS use either statistical measures or computed thresholds on feature sets but are ineffective for present day attacks. It is because they suffer from high rate of false positive and false negative alerts, where the high rate of false positive alerts mean that could unnecessarily alert even when there are no attacks happened, while high rate of false negative alerts mean that NIDS could fail to detect attacks more frequently.

Network intrusion detection operated using sets of rule and code signatures created by experts but it is time consuming and can only be created if the attack method that was chosen has been used at least once. In order to solve these issues, machine learning algorithms are being used into NIDS. The existing intrusion detection has been more effective with the development of machine learning recently even though there are still problems with low detection accuracy due to the instability of machine learning algorithm itself.

### 1.4.2 Machine Learning Techniques vs Deep Learning Approaches

Machine learning is a branch of artificial intelligence that adapted to the new environment which allows programs to find and learn the patterns within data. Machine learning is divided into three sub-domains, which are supervised, unsupervised and reinforcement learning as shown in Figure 1.5.

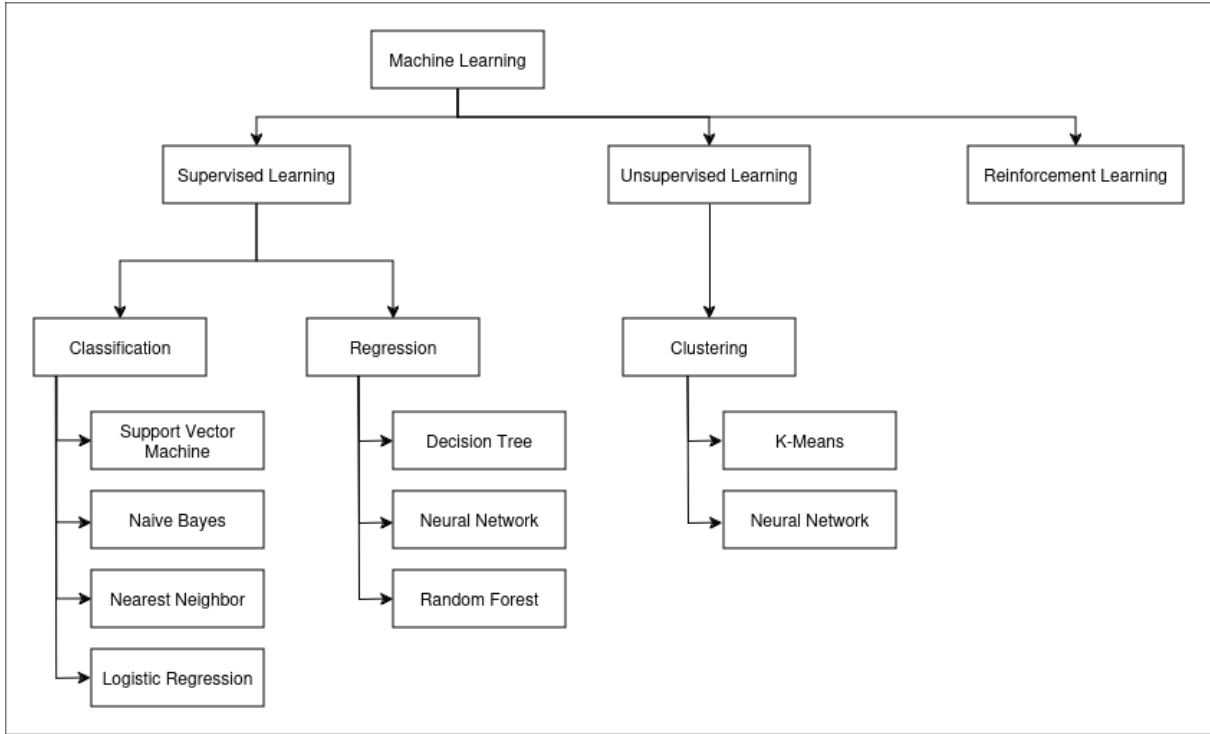


Figure 1.5: Machine Learning methods.

The Deep Learning is a machine learning approach that consists of multiple-level layers that is capable of running processes simultaneously and high-level features are produced from the low-level ones. Thus, deep learning takes action by forming its own features without using human power. Similar with machine learning, deep learning is also divided into several sub-domains, which are supervised and unsupervised learning as shown in Figure 1.6.

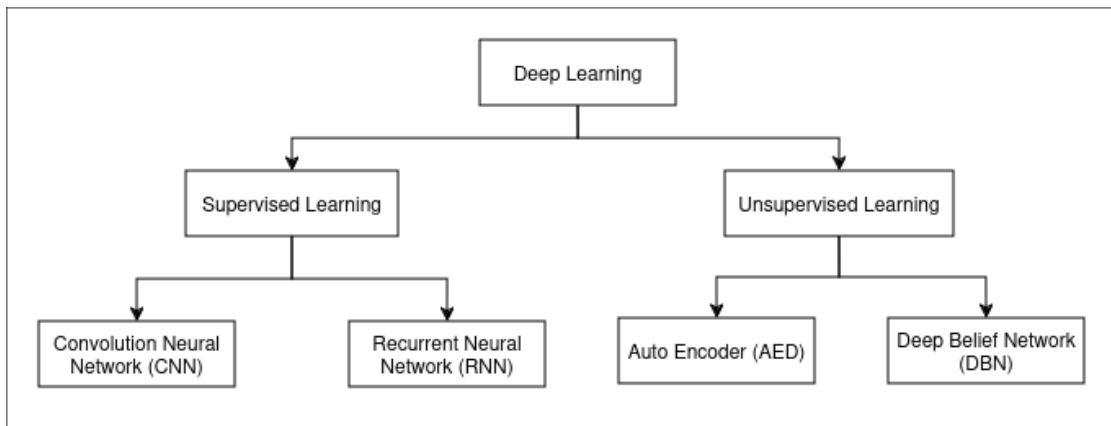


Figure 1.6: Deep Learning methods.

Convolutional Neural Network (CNN) is a special type of artificial neural network (ANN) and also the most frequently used deep learning method. CNN are made up of neurons that capable to learn biases and weights and it also process data that comes in multiple arrays and removes the need for manual feature extraction.

ML- or DL-based models must be trained on a number of samples. For this training process to be effective, large datasets are required. Resolving the relation between the ML model size and the required amount of data has been a prominent research area in the past decade. This problem affects the development of robust and up-to-date IDS. Datasets are often depicted as the bottleneck for developing robust ML models due to the following reasons:

1. gathering large realistic datasets is a complex task and requires a lot of manual labour;
2. using synthetic or deprecated datasets makes it difficult for the developed model to fit in real-life deployments;
3. training classical ML models with small datasets exposes the models to over-fitting problems;
4. continuous generation of datasets to cope with emerging attacks.

From these issues came the idea of using the One-Shot learning. It focuses on learning new classes from only one or few examples. One of the most popular ML techniques, and a building block of other ML models including Siamese Netowkes, is Artificial Neural Network (ANN). ANN is inspired by the human brain, thus its building block is the artificial neurons. An artificial neuron is composed of three elements: (a) input, (b) output and (c) activation function. Typically, an ANN is composed of an input layer, an output layer, and zero or more hidden layers. Each of these layers is composed of multiple neurons. Neurons in each layer are connected to the ones in the following layer using connections called ‘weights’. An ANN is trained (the weights are adjusted) to best minimise the loss. Once the ANN is trained, the input neurons values are propagated using weights/connections and activation functions to correspond to the desired output. A Siamese Network is composed of two identical ANN called "twin networks". These twin networks share the same weights and they train simultaneously. This network, unlike other ML techniques, is trained to decide whether a given pair is similar or not. The output is the degree of similarity which can also be squashed to a binary similar/dissimilar output. Siamese Network usage has advanced in various domains, for example for image processing usage. Although Image and Video processing has been the prominent domain, Siamese Networks are used in the medical domain and Natural Language Processing (NLP) domain [14].



### 1.4.3 Overview of Existing Datasets

In recent years, the field of IoT security has seen the emergence of numerous datasets, each with its own set of advantages and disadvantages. As the pool of undetected vulnerabilities and threats continues to expand, there is a growing emphasis by researchers on datasets related to IoT. IoT devices' performance, security, and relevance, whether under typical or anomalous conditions, are assessed through data collection in either simulated or genuine environments. Consequently, the dataset's quality is pivotal in creating a robust model for real-world intrusion detection, especially for the problems mentioned earlier. While many studies predominantly use datasets like the KDD Cup 1999, NSL-KDD, UNSW-NB15 and TON\_IoT other datasets can also serve the purpose of cybersecurity intrusion detection. This section delves into publicly available datasets recommended for intrusion detection systems (IDS) within IoT scenarios [15].

#### **KDDCUP99**

The KDDCup99 dataset, used in the Third International Knowledge Discovery and Data Mining Tools Competition [16], is designed to differentiate between "malicious" and "benign" network connections for the development of a robust NIDS. Derived from the DARPA dataset, KDDCup99 comprises approximately 4.9 million connection records, each represented by 41 features. Each connection is categorized as either an attack or normal. The dataset encompasses various security attacks, including DoS, U2R, R2L and Probing Attacks.

#### **NSL-KDD**

Introduced by Tavallae et al. [16], the NSL-KDD dataset is a refined version of the KDDCup99 dataset. Retaining the same features as KDDCup99, NSL-KDD was curated by removing redundant and repetitive records and optimizing the dataset size. This dataset features 41 attributes along with a class label. The class label is segmented into 21 categories, further grouped into four primary attack types: probe, U2R, R2L and DoS.

#### **UNSW-NB15**

Developed by the Australian Centre for Cyber Security in their Cyber-Range Lab using the IXIA PerfectStorm tool, the UNSW-NB15 dataset [17] aims to capture a blend of genuine normative behaviors with synthetically generated contemporary cyber attacks. The dataset comprises 2,540,044 records, of which 2,218,761 are benign, and 321, 283 are malicious. The dataset encompasses nine distinct attack types: backdoors, fuzzers, analysis, shellcode, DoS, exploits, reconnaissance, worms and generic.

### **CICIDS2017**

The CICIDS2017 dataset, curated by the same institution as Sharafaldin et al. [18], is a contemporary collection of various attack scenarios. The dataset was created with genuine user-generated background traffic from the B-Profile system. It captures diverse types of attacks such as DDoS, DoS, Heartbleed, Web Attack, Infiltration, Botnet, Brute Force SSH, and Brute Force FTP. The dataset employs the CICFlowMeter tool to extract eighty distinct network flow characteristics from the captured traffic.

### **BoT-IoT**

The BoT-IoT dataset, crafted by Koroniotis et al. [19] at UNSW Canberra Cyber Range Lab, encompasses legitimate and malicious traffic data from simulated IoT devices. This testbed includes network devices, notably the pfSense firewall, attack and target virtual machines (VMs), and simulated IoT devices operating within VMs connected to the AWS IoT hub. With 73,370,443 network traffic records, the dataset portrays a smart house environment containing a weather station, smart fridge, smart thermostat, remotely operated garage door, and smart lights. This dataset catalogs various attacks, including keylogging, data exfiltration, OS and service scans and DoS and DDoS attacks.

### **DS2SoS**

Introduced by Pahl et al. [20], DS2SoS is a next-generation, open-source IIoT security dataset tailored for research. It aids in evaluating the efficacy of ML/DL-driven cybersecurity algorithms, especially in smart factory and city contexts. The dataset holds 357,952 samples, split into 10,017 anomalies and 347,935 regular data points. It features 13 distinct attributes and seven categories of attacks, including denial of service, malicious operation, incorrect setup, espionage, scans and data-type probing incursions.

### **CSE-CIC-IDS2018**

The CSE-CIC-IDS2018 dataset [21] was devised as a superior replacement for existing datasets limiting IDS/NIDS experimental evaluations. This dataset highlights seven diverse attack scenarios: brute force, heartbleed, botnets, DDoS web assaults, and local network infiltrations. The hypothetical target organization consists of 5 departments, 30 servers, and 420 hosts, summing up to 50 nodes in its attack blueprint. The authors extracted 80 distinctive features from computer logs and network traffic using CICFlowMeter-V3.

### **CICDDoS2019**

Sharafaldin et al. [22] curated the CICDDoS2019 dataset, focusing on DDoS attacks and leveraging the publicly accessible CICFlowMeter tool from the Canadian Institute for Cy-

bersecurity derived 80 network traffic attributes for all benign and malicious flows. This research delves into contemporary attacks executed via TCP/UDP application-layer protocols and introduces two novel attack categories: reflection-based DDoS and exploitation-based. In both, attackers exploit legitimate third-party components to obfuscate their identity. The dataset simulates user behaviors for 25 individuals across protocols like SSH, HTTP, HTTPS, FTP and email.

### **UNSW-SOSR2019**

The UNSW-SOSR2019 dataset, sourced by the University of New South Wales security lab using the tcpdump tool [23], archives traffic from 10 distinct IoT devices. This dataset chronicles benign and malicious traffic, detailing attacks such as ARP spoofing, Fraggle (UDP flooding), TCP SYN flooding, and Ping of Death. Reflective attack types, including SNMP, TCP SYN, SSDP and Smurf are also cataloged.

### **ToN-IoT**

Developed jointly by the Cyber Range and UNSW Canberra IoT Labs [24], the ToN-IoT dataset amalgamates data from diverse sources within a comprehensive IIoT system. This includes network traffic, Linux and Windows OS logs, and telemetry from connected gadgets. The dataset identifies many attacks: ransomware, password attacks, scans, DoS, DDoS, XSS, data injection, backdoors, and MITM attacks, to name a few. Boasting 22,339,021 records, the dataset features 44 attributes grouped into four service-profile-based categories detailing connection, user activities (e.g., DNS, HTTP, SSL), statistics and breach characteristics.

### **IoT-23 dataset**

The IoT-23 dataset, curated by the Stratosphere Laboratory of the Czech Technical University (CTU) [25] provides researchers with a comprehensive collection of genuine IoT data, including benign and malicious activities. It encompasses three benign and 20 malicious actions and 20 network operation models that mimic IoT device scenarios. The dataset comprises 325,307,990 records distributed across nine attack categories: DDoS, HeartBeat, Mirai, Okiru, Torii, C&C, PartOfAHorizontal, PortScan and FileDownload.

### **MQTT-IoT-IDS2020**

Hindy et al. developed the MQTT-IoT-IDS2020 dataset, which focuses on conventional and brute-force attacks targeting the MQTT networking framework [26]. The network structure includes 12 MQTT sensors, an MQTT broker, a camera feed replication mechanism, and an intrusion detection system. The four primary attack types presented are Sparta SSH brute-force, UDP scan, Aggressive scan and MQTT brute-force.

## Edge-IIoT

The Edge-IIoT dataset, introduced by the authors in Ref. [27], is designed to facilitate intrusion detection research. It enables the evaluation of federated deep learning and centralized intrusion detection systems using universally accepted metrics. The dataset describes 14 attacks associated with IoT and IIoT protocols, further classified into five threat categories: information gathering, DoS and DDoS attacks, injection attacks, malware-based attacks, and man-in-the-middle attacks. It encompasses 1176 features, with 61 of them being highly correlated. The dataset documents 20,780,120 attack-related records, of which 11,050, 411 are benign, and 9,729,709 are malicious. Traffic predictability and detection efficacy were assessed across cyber-threats using binary, 6-category and 15-category classifications. The evaluation employed classifiers like RF, SVM, kNN and DNN.

The following table (Table 1.1) resume the differences between datasets described:

Dataset	Advantages	Disadvantages
<b>KDDCup99</b> [16]	Large dataset with approximately 4.9 million records. Includes various attack types: DoS, U2R, R2L, Probing.	Outdated; based on data from 1999. Contains redundant and duplicate records. Not specific to IoT environments.
<b>NSL-KDD</b> [16]	Improved version of KDDCup99 with redundant records removed. Balanced dataset size for better evaluation.	Still based on outdated data. Lacks IoT-specific features and modern attack types.
<b>UNSW-NB15</b> [17]	Contains contemporary attack types. Mix of real and synthetic data. Large dataset with over 2.5 million records.	Synthetic attacks may not capture real-world complexities. Not tailored specifically for IoT devices.
<b>CICIDS2017</b> [18]	Includes recent and diverse attack scenarios. Realistic user behavior. Extracted 80 network flow features.	Focused on general network traffic, not IoT-specific. Does not cover IoT protocols like MQTT or CoAP.
<b>BoT-IoT</b> [19]	Simulates IoT devices and environments. Encompasses various IoT-relevant attacks.	Uses simulated devices within virtual machines. May lack nuances of traffic from actual IoT devices.
<b>DS2OS</b> [20]	Tailored for Industrial IoT (IIoT) security research. Open-source dataset.	Limited size with fewer anomalies. Focused on IIoT; may not generalize to consumer IoT devices.

---

# 1. STATE OF THE ART

<b>CSE-CIC-IDS2018</b> [21]	Addresses limitations of older datasets. Seven diverse attack scenarios. Extracts 80 features from network traffic and logs.	Not specific to IoT environments. May not include IoT-specific vulnerabilities or attacks.
<b>CICDDoS2019</b> [22]	Focuses on DDoS attacks, including new types. Uses standardized tools for feature extraction.	Limited to DDoS attacks. Not specifically designed for IoT devices.
<b>UNSW-SOSR2019</b> [23]	Captures traffic from real IoT devices. Includes both benign and malicious traffic.	Dataset size may be limited. May not cover a comprehensive range of IoT threats.
<b>ToN-IoT</b> [24]	Combines data from multiple IIoT resources. Covers a wide array of attacks. Large dataset with over 22 million records.	Synthetic environment may not reflect real-world conditions. High complexity with many features.
<b>IoT-23</b> [25]	Provides real IoT data with both benign and malicious activities. Includes 20 network operation models.	Very large dataset size can be cumbersome. May lack certain protocol-specific attacks like MQTT.
<b>MQTT-IoT-IDS2020</b> [26]	Focused specifically on attacks targeting MQTT. Includes real-device test cases.	Limited to MQTT and certain attack types. Smaller dataset size with less diversity.
<b>Edge-IIoT</b> [27]	Facilitates research on intrusion detection. Covers 14 attacks associated with IoT protocols. Large dataset with over 20 million records.	High dimensionality complicates data processing. Focused on IIoT; may not represent consumer IoT devices.

Table 1.1: Comparison of IoT Attack Datasets

The necessity to create a new dataset emerged from significant gaps in existing IoT security datasets. Many available datasets rely on simulated environments or virtual devices, which do not accurately capture the complexities and nuances of real-world IoT device behavior. Additionally, most lack specific attacks targeting the MQTT protocol, a primary communication method for IoT devices. This absence limits the effectiveness of intrusion detection systems (IDS) developed using these datasets, as they may not adequately detect or mitigate MQTT-based threats.

To address these shortcomings, a new dataset was developed using real IoT devices to ensure the data reflects genuine device behaviors and inherent vulnerabilities. The dataset is composed in two versions to highlight the influence of MQTT traffic on IoT security. The first version includes four classes: benign, DoS, reconnaissance and brute-force attacks. The second version adds a fifth class, MQTT attack, resulting in the classes: benign, DoS, reconnaissance, brute-force and MQTT attack. Both versions consist of 31 features that effectively capture the nuances of IoT network traffic and attack patterns. The total size of dataset is 51.838.560 samples.

The main advantage of this new dataset is its enhanced realism and relevance. By in-

cluding both traditional network attacks and MQTT specific attacks, it provides a more comprehensive threat model for IoT environments. This dual-version approach allows for comparative analysis, offering valuable insights into how MQTT communication affects the security posture of IoT systems. However, the dataset's relatively small size is a limitation, stemming from the limited data traffic that can be captured from real devices in practical scenarios. This smaller size may impact the statistical significance and generalizability of findings derived from it.

Despite this limitation, the use of Siamese networks can overcome the challenges posed by the smaller dataset. Siamese networks are particularly well-suited for scenarios with limited data because they require fewer samples to train effectively. They operate by learning to differentiate between pairs of inputs, focusing on the similarity or dissimilarity between them. This approach allows the model to make better use of the available data by generating numerous pairs from the existing samples, effectively expanding the training set without the need for additional data. Consequently, even with a smaller dataset, a Siamese network can achieve robust performance in detecting and classifying attacks within IoT systems.

In conclusion, while the dataset's size is a constraint, the combination of high-quality, realistic data and advanced modeling techniques like Siamese networks can mitigate this challenge. The new dataset provides a valuable resource for advancing IoT security research and developing more effective IDS solutions tailored to the unique challenges of modern IoT environments. By leveraging Siamese networks, researchers can maximize the utility of the dataset, contributing to the creation of robust security measures that protect against both traditional and MQTT specific threats in real-world IoT deployments.

### 1.4.4 Challenges in Implementing IDS on Embedded Systems

Embedded systems are becoming increasingly deployed in critical situations. For example, modern cars are equipped with embedded devices that control different parts of the car such as Anti lock Break System (ABS), and Adaptive Cruise Control (ACC). These embedded devices carry out critical tasks and hence, are potential targets for malicious users. They are also equipped with data and network interfaces which result in an increased attack surface.

Intrusion Detection Systems (IDS) are widely deployed in general purpose computer systems to protect them from attacks. Based on the critical nature of the applications of embedded devices, building IDSes for these systems is a necessity. However, embedded devices have characteristics that make building IDSes for them challenging as follows:

- *Limited memory capacity*: an important component of an IDS is an artificial intelligent model that may occupy a large space in memory. However, many embedded devices have limited memory capacity and this makes a body of existing techniques inapplicable to them.

- *Large scale deployment*: this implies that the security mechanism should not have false positives. False positives occur when no actual attack has happened but an attack is reported by the IDS. Even a small rate of false positive for systems deployed on a large scale aggregates quickly. Further, these systems are deployed in mission critical settings where shutting them down on a false alarm is not a viable option.

Existing techniques for building IDSs for general computers are not suitable for embedded devices. For example, statistical techniques have false positives that make them impractical for embedded devices that are deployed on a large scale. Techniques based on static analysis do not have false positives as they build a model by conservatively taking into account all possible code behavior. However, the size of this model may exceed the limited memory of embedded devices, and these techniques offer no systematic way to reduce the size of the model. For instance, Wagner et. al. [28] and Giffin et. al. [29] use system calls to build a model of the system. Reducing the size of the model by randomly removing some of the system calls results in a model that may no longer provide any guarantees on the coverage of the system behavior [30].

# Chapter 2

## My Cloud-Based IoT Architecture

In chapter 2, the creation of a Cloud-based IoT architecture for a domotic home will be explored, focusing on the design and implementation of a smart door that opens and closes exclusively via a fingerprint reader. This advanced system not only ensures that only authorized users can gain access, but also alerts the owner in case of breaches due to too many failed access attempts. In addition to the fingerprint reader and alarm sensor, the system also includes a person detection sensor. This sensor notifies the user when a person is detected in front of the door and signals when they move away. This additional functionality not only increases the level of security, but also gives the owner greater awareness of what is happening outside their home. All information collected by the sensor is stored in the cloud and can be accessed through the dedicated application, allowing real-time monitoring and the ability to react quickly to suspicious situations. In the course of the chapter, I will detail the design and construction processes of the architecture, the hardware elements chosen, the software used and the logic adopted to realize this innovative system.

### 2.1 Architecture Design

This cloud-based IoT architecture represents a home automation door that opens via a fingerprint reader. The user has a maximum of three attempts to gain access, after which an alarm is triggered and the door is locked. In addition to the alarm system, there is also a sensor that detects the presence of a person in the vicinity of the door and notifies when that person moves away, in order to guarantee a higher level of security. All this information and notifications are stored in the cloud and can be accessed by the user via a dedicated application.



### 2.1.1 Hardware components

The devices involved in the architecture are three embedded devices (ESP32, Arduino Uno WiFi Rev2 and ESP8266) and a Raspberry Pi that acts as the main device, providing connectivity and managing communication between the various components. The ESP32 is responsible for controlling the fingerprint reader, the Arduino manages the person detection sensor and takes care of the alarm system. The Raspberry Pi coordinates the entire system, ensuring data is stored in the cloud and allowing the user to access information and control the door via the dedicated application.

The following tables (Table 2.1 to Table 2.4) showing the names of the devices used and their technical descriptions.


TTGO LoRa32-OLED	Datasheet
	<p>The ESP32 is a single chip manufactured with TSMC's ultra-low-power 40nm technology, which integrates Wi-Fi and Bluetooth functionality at 2.4 GHz. It is recognised as the most integrated Wi-Fi + Bluetooth solution in the industry, requiring less than 10 external components. It supports various protocols including TCP/IP, 802.11 b/g/n/e/i and Bluetooth v4.2 BR/EDR and BLE. The chip is equipped with two 32-bit Xtensa® LX6 dual-core microprocessors, capable of reaching up to 600 DMIPS. It offers numerous peripheral interfaces such as ADCs, DACs, touch and temperature sensors, as well as SPI, I2S, I2C and UART interfaces. ESP32 supports hardware acceleration for cryptographic algorithms such as AES, HASH, RSA and ECC, providing enhanced application security. It operates with a supply voltage of 2.2 V to 3.6 V and includes power-saving features. It has 16 MB of flash memory and 520 KB of SRAM. [31].</p>

Table 2.1: Datasheet TTGO LoRa32-OLED board.

## 2. MY CLOUD-BASED IOT ARCHITECTURE

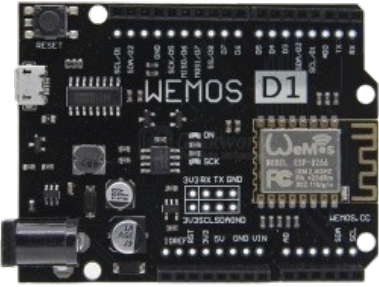
WeMos D1 ESP8266 WiFi	Datasheet
	<p>The WeMos D1 R2 board is a development board that uses the ESP8266EX microchip and offers compatibility with the Arduino IDE and NodeMCU. Operating at 3.3V, the board has 11 digital I/O pins, each of which supports interrupt/pwm/I2C/one-wire functionality, with the exception of pin D0. A single analogue input pin allows readings up to a maximum of 3.2V. The board integrates a 4MB flash memory and a switching power supply that accepts input voltages from 9V to 12V, providing an output of 5V at 1A maximum. It is important to remember that the WeMos D1 R2 operates at 3.3V: a logic level converter must be used to interface it with 5V sensors or digital devices [32].</p>

Table 2.2: Datasheet WeMos D1 ESP8266 WiFi board.

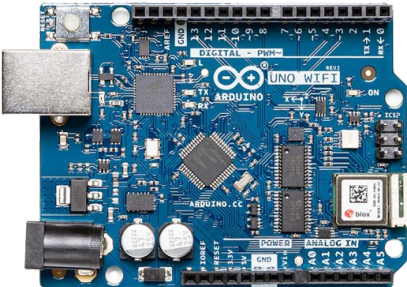
Arduino Uno WiFi Rev2	Datasheet
	<p>The Arduino UNO WiFi Rev2 board is based on the ATmega32U4 microcontroller and offers WiFi and Bluetooth connectivity via the NINA W102 module. Powered at 5V, the board supports a maximum current of 40 mA per pin, with a recommendation of 20 mA for optimum use. The maximum current allowed for the entire package is 200 mA. The board has 20 digital pins, 7 of which can be used as PWM outputs and 6 as analogue input pins. I2C and SPI communication is handled via dedicated pins. The UNO WiFi Rev2 also includes a reset pin, an AREF pin and an IOREF pin for greater flexibility in use. [33].</p>

Table 2.3: Datasheet Arduino Uno WiFi Rev2 board.


Raspberry Pi 3 Model B	Datasheet
	<p>The Raspberry Pi 3 Model B has a Broadcom BCM2387 chipset, with a quad-core ARM Cortex-A53 1.2GHz processor. The BCM2837 chip also integrates 802.11 b/g/n Wireless LAN and Bluetooth 4.1 connectivity, which includes both Bluetooth Classic and Bluetooth Low Energy (LE), offering full connectivity. Graphics capabilities are handled by the Dual Core VideoCore IV® multimedia coprocessor, which supports OpenGL ES 2.0, hardware-accelerated OpenVG and H.264 high-profile decoding at 1080p30. This graphics coprocessor can handle an impressive throughput of 1 Gpixel/s, 1.5 Gtexel/s or 24 GFLOPs with texture filtering and DMA infrastructure. As far as memory is concerned, the Raspberry Pi 3 Model B is equipped with 1 GB of LPDDR2 RAM. The operating system, which can be a Linux distribution or Windows 10 IoT, is booted from a Micro SD card. Powered via a 5V and 2.5A Micro USB connector, the Raspberry Pi 3 Model B offers a wide range of connectivity options, including a 10/100 BaseT Ethernet port, HDMI video output (rev 1.3 and 1.4) and RCA composite output (PAL and NTSC). Audio output is available via both a 3.5 mm jack and HDMI. There are also four USB 2.0 ports for connecting peripherals. The Raspberry Pi 3 Model B has a 40-pin (2x20 strip) GPIO connector with 2.54 mm (100 mil) pitch, which provides 27 GPIO pins, as well as +3.3 V, +5 V and GND power lines. A 15-pin Camera Serial Interface (CSI) connector allows connection of camera modules, while a 15-way Display Serial Interface (DSI) connector with two data lines and a clock line allows connection of displays. An insertion/extraction Micro SDIO memory card slot completes the expansion options [34].</p>

Table 2.4: Datasheet Raspberry Pi 3 Model B board.

---

## 2. MY CLOUD-BASED IOT ARCHITECTURE

---

The following tables (Table 2.5 to Table 2.7) showing the names of the sensors used and their technical descriptions.

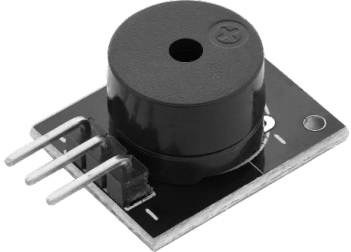
Buzzer 3 Pins	Datasheet
	The KY-006 passive piezoelectric buzzer emits a sound when a voltage is passed through it. By varying the frequency of the PWM signal applied to the buzzer, different sounds can be generated. The buzzer can be controlled using a microcontroller such as Arduino or Raspberry Pi [35].

Table 2.5: Datasheet Buzzer 3 Pins sensor.


Buzzer 3 Pins	Datasheet
	The fingerprint module can store up to 880 fingerprint templates in its flash library. It uses an optical sensor to capture the fingerprint image and a Synochip DSP to process it. The module offers different matching modes, including 1:1 and 1:N, and supports different security levels. The average search time for a 1:880 match is less than 1 second [36].

Table 2.6: Datasheet Buzzer 3 Pins sensor.


Buzzer 3 Pins	Datasheet
	The HC-SR501 PIR motion sensor uses infrared technology to detect movement. When a warm body, such as a person, enters its field of view, the sensor emits a high signal. The sensor has a detection range of 7 metres and a detection angle of 110 degrees. Both the detection distance and the delay time can be adjusted [37].

Table 2.7: Datasheet Buzzer 3 Pins sensor.

## 2. MY CLOUD-BASED IOT ARCHITECTURE

In the Figure 2.1 we can see the logic diagram and the configuration of the implemented Cloud-based IoT architecture:

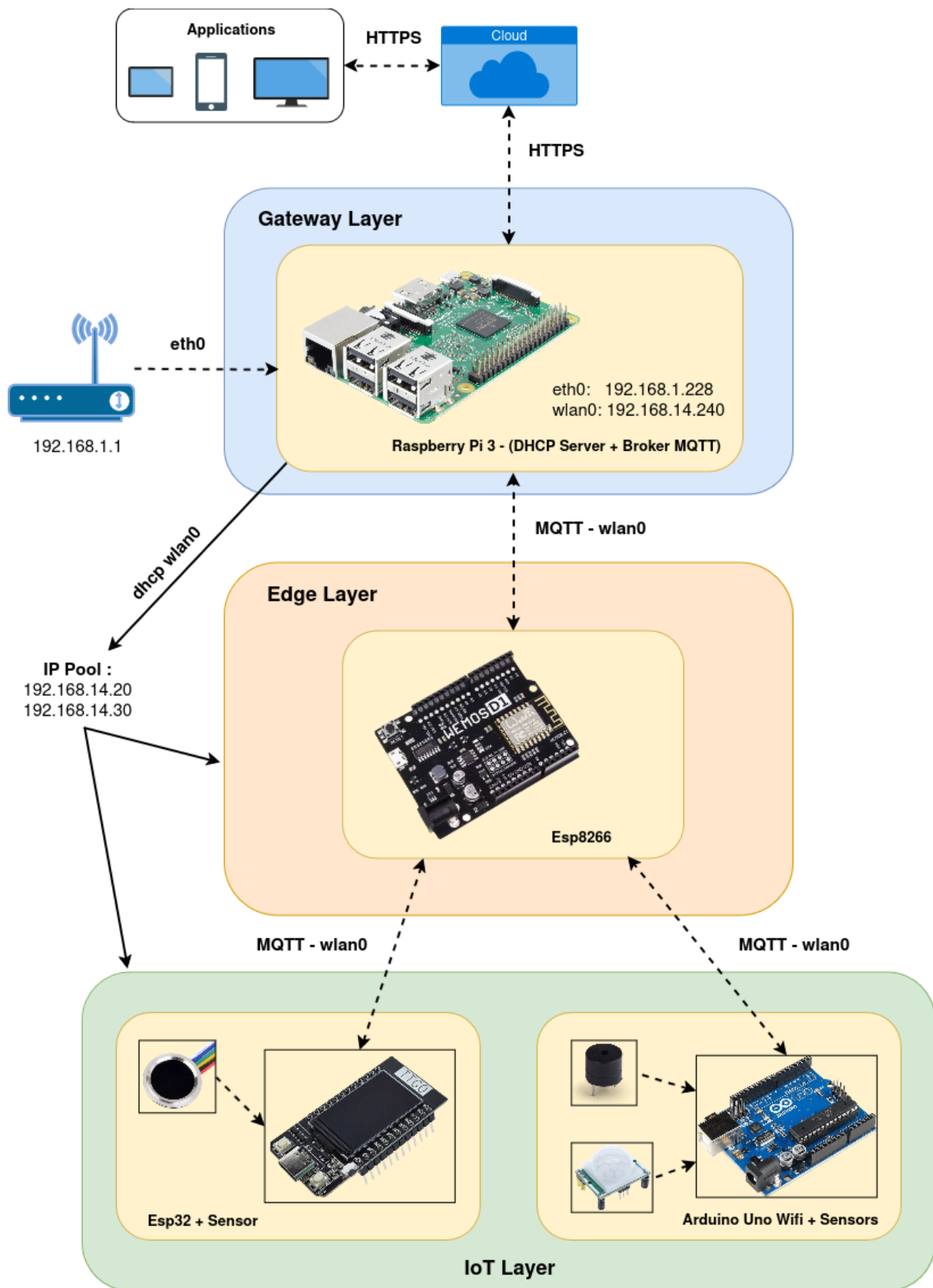


Figure 2.1: My Cloud-based IoT Architecture.

The logic of the architecture is divided into several levels:

- *IoT Layer*: in this level there is the ESP32 board with fingerprint sensor and Arduino Uno WiFi Rev2 with proximity sensors and buzzer. The objective of the ESP32 and Arduino Uno WiFi Rev2 is to collect data from the corresponding sensors and send it to the Edge layer communicating via the MQTT protocol.
- *Edge Layer*: here ESP8266 receives the information sent by ESP32 and Arduino via MQTT protocol, filters it and sends it to the Raspberry Pi or to ESP32 and Arduino with the MQTT protocol;
- *Gateway Layer*: in this layer the Raspberry Pi also collects the information transmitted from the ESP8266, updates a CSV file on Google Drive, it sends data to user and to ESP8266. The Raspberry Pi is networked via a wired connection to the modem, while, via the WLAN, it performs the functions of DHCP server (to provide IP addresses to the boards) and MQTT broker. The WLAN interface is configured so that the Raspberry Pi acts as an access point to allow communication between all the boards. Configuration of the WLAN network was done by setting a static IP to the Raspberry Pi on a network other than the Ethernet network, thus ensuring a separation between the two networks and optimizing network traffic management.

## 2.2 Network Creation and Device Communication

This section will illustrate the creation of the network and its configuration using the Raspberry Pi. I will show two types of network configuration, one secure and the other insecure.

The Raspberry Pi plays a crucial role, acting as both Access Point and MQTT broker. The Access Point was configured via `hostapd`, allowing the boards to connect and communicate with each other. Since the boards do not communicate with the outside world, it was not necessary to implement IP forwarding; they only communicate with the Raspberry Pi and with each other. IP addresses are provided by the Raspberry Pi, which acts as a DHCP server via `dnsmasq`. To begin with, a static IP was assigned to the Raspberry Pi using `dhcpcd`. Next, a pool of IP addresses was created and randomly assigned to each connected device. Finally, the Mosquitto broker was installed on the Raspberry Pi to manage MQTT communication between the boards.

### 2.2.1 Initial configurations

For this configuration, the Raspberry Pi must have *Debian version Bullseye* as OS in order to use `dhcpcd` as the network interface. In fact, in later versions `dhcpcd` is replaced with the *Network Manager*.

After installation of the Raspberry Pi OS [38], configuration of the Raspberry Pi was carried out by remotely accessing the board via the SSH service, entering the correct username and password credentials. Once logged in, the first thing to do is to update the system:

```
sudo apt update
sudo apt upgrade -y
```

### Create *watchdog\_raspi.service*

The *watchdog\_raspi* application is a tool developed in Python, designed to operate on Raspberry Pi devices, with the aim of facilitating the synchronization of user operations performed by the device via its user-application. At the heart of this application is a CSV file stored on Google Drive, which serves as a centralized repository for activity logs.

When a user interacts with the client application to perform an action or a board sends information, all that is immediately recorded in the CSV file. This ensures a persistent and chronological tracking of all interactions and operations performed. Each line of the CSV contains details such as the timestamp of the action and the type of operation.

Integration with Google Drive takes place via the API provided by Google, allowing real-time synchronization between the Raspberry Pi and the CSV file stored in the cloud. The application implements secure authentication mechanisms, using a service account: the credentials file (*iot-raspi.json*) is uploaded and the access tokens needed to interact with the Google Drive API are obtained. If an error occurs during authentication, an error message is recorded and printed.

The main flow manages the entire application flow:

- *Authentication*: attempts to authenticate with Google Drive. If it fails, it retries every 5 seconds.
- *CSV File Verification*: waits for the *iot.csv* file to be present on Google Drive, retrying every 5 seconds if not found immediately.
- *MQTT Connection and Subscription*: establishes connection to the MQTT broker and subscribes to the specified topic, waiting for both operations to complete successfully.
- *Continuous Monitoring*: enters an infinite loop in which it constantly checks whether the CSV file on Google Drive has been modified. If a change is detected, it downloads the updated file, processes the last recorded operation, and Raspberry Pi sends the informations via MQTT. Then, handles any disconnections from the MQTT broker by attempting to reconnect and re-establish the subscription.



---

## 2. MY CLOUD-BASED IOT ARCHITECTURE

---

To ensure that the Python application starts automatically whenever the system is turned on, a *systemd* service must be created and configured:

```
sudo nano /etc/systemd/system/watchdog_raspi.service
```

Within this file, details on how to run the Python script, the working directory, automatic restart policies and other relevant settings are specified:

```
[Unit]
Description=Python script that allows you to keep a csv file updated
After=network.target

[Service]
ExecStart=python /home/alberto/watchdog_raspi/cloud-raspy.py
WorkingDirectory=/home/alberto/watchdog_raspi/
StandardOutput=inherit
StandardError=inherit
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

After saving and closing the configuration file, I reload the systemd daemon configuration to apply the changes, enable the service to start automatically at boot and start the service immediately to verify that it is working properly:

```
sudo systemctl daemon-reload
sudo systemctl enable watchdog_raspi.service
sudo systemctl start watchdog_raspi.service
```

Using commands such as `systemctl status`, I can monitor the status of the service and ensure that it is up and running. Finally, by performing a system reboot, I can confirm that the *watchdog\_raspi* application starts automatically as expected.

### App Client

The client application manages operations of users saving them on a CSV file in Google Drive using the *Google Drive API* for managing the file and the *Rich* library for visually presenting the information to the user. The structure of the programme provides several functions that manage authentication, reading and writing data in the CSV file to Google Drive. Authentication is realized through the use of the `google_auth_oauthlib` library, which allows the user to authorize the application to interact with their Google Drive. Once authentication has been performed, the `authenticate` function returns an authenticated service that allows operations to be performed on Google Drive resources.



```
SCOPES = ['https://www.googleapis.com/auth/drive.file']
CREDENTIALS_FILE = 'iot_app.json'
FILENAME = 'iot.csv'
MIMETYPE = 'text/csv'
HEADERS = ['time', 'operation']

console = Console()

def authenticate():
    flow = InstalledAppFlow.from_client_secrets_file(CREDENTIALS_FILE, SCOPES)
    creds = flow.run_local_server(port=0)
    return build('drive', 'v3', credentials=creds)
```

The CSV file used by the application is managed through the functions `get_file_id` and `check_and_create_csv`. The first function has the task of searching for the `iot.csv` file on Google Drive, returning its ID if it exists, or a *None* value if the file does not exist.

```
def get_file_id(service):
    try:
        results = service.files().list(q=f"name='{FILENAME}'", fields="files(id,
↪ name)").execute()
        items = results.get('files', [])
        if not items:
            return None
        return items[0]['id']
    except HttpError as error:
        console.print(f"[red]Errore durante la ricerca del file CSV: {error}[/red]")
        exit(1)
    except Exception as e:
        console.print(f"[red]Errore imprevisto durante la ricerca del file CSV: {e}[/red]")
        exit(1)
```

If the file does not exist, the `check_and_create_csv` function creates a new one with the default headers and saves it to Google Drive.

```
def check_and_create_csv(service):
    file_id = get_file_id(service)
    if not file_id:
        console.print(f"[yellow]{FILENAME} non esiste, lo creo.[/yellow]")
        df = pd.DataFrame(columns=HEADERS)
        buffer = BytesIO()
        df.to_csv(buffer, index=False)
        buffer.seek(0)
        file_metadata = {'name': FILENAME}
        media = MediaIoBaseUpload(buffer, mimetype=MIMETYPE, resumable=True)
        try:
            service.files().create(body=file_metadata, media_body=media,
↪ fields='id').execute()
            console.print(f"[green]{FILENAME} creato con successo.[/green]")
        except HttpError as error:
            console.print(f"[red]Errore durante la creazione del file CSV: {error}[/red]")
            exit(1)
        except Exception as e:
```

```
        console.print(f"[red]Errore imprevisto durante la creazione del file CSV:
        ↳ {e}[/red]")
        exit(1)

    else:
        console.print(f"[green]{FILENAME} già esiste.[/green]")
```

The functions `read_csv_from_drive` and `upload_csv_to_drive` handle the reading and writing of the CSV file to and from Google Drive respectively. The reading is done using the `get_media` method of the API, while the upload is handled with the `update` method, which allows the contents of the file to be updated.

```
def read_csv_from_drive(service):
    file_id = get_file_id(service)
    if not file_id:
        console.print("[red]Errore: file non trovato.[/red]")
        return None, None
    try:
        request = service.files().get_media(fileId=file_id)
        buffer = BytesIO()
        downloader = MediaIoBaseDownload(buffer, request)
        done = False
        while not done:
            status, done = downloader.next_chunk()
            buffer.seek(0)
            return pd.read_csv(buffer), file_id
    except HttpError as error:
        console.print(f"[red]Errore durante il download del file CSV: {error}[/red]")
        return None, None
    except Exception as e:
        console.print(f"[red]Errore imprevisto durante il download del file CSV: {e}[/red]")
        return None, None

def upload_csv_to_drive(service, df, file_id):
    try:
        buffer = BytesIO()
        df.to_csv(buffer, index=False)
        buffer.seek(0)
        media = MediaIoBaseUpload(buffer, mimetype=MIMETYPE, resumable=True)
        service.files().update(fileId=file_id, media_body=media).execute()
    except HttpError as error:
        console.print(f"[red]Errore durante l'upload del file CSV: {error}[/red]")
        exit(1)
    except Exception as e:
        console.print(f"[red]Errore imprevisto durante l'upload del file CSV: {e}[/red]")
        exit(1)
```

The application provides interaction with the user through a command-line interface that uses the Rich library to enhance the user experience and present data more clearly. The user can view the operations recorded in the file, lock or open a door, trigger an alarm, delete rows based on a specific period, or empty the contents of the file completely. These operations are managed through the use of functions such as `log_operation`, `delete_rows` and `clear_file`.

## 2. MY CLOUD-BASED IOT ARCHITECTURE

---

```
def log_operation(service, operation):
    def operation_logic():
        now_1 = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        df, file_id = read_csv_from_drive(service)
        if df is None:
            return None
        new_row = pd.DataFrame({'time': [now_1], 'operation': [operation]})
        df = pd.concat([df, new_row], ignore_index=True)
        upload_csv_to_drive(service, df, file_id)
        return now_1

    now = progress_bar(operation_logic)
    if now:
        console.print(f"[green]\nOperazione '{operation}' registrata alle {now}[/green]")

def delete_rows(service, period):
    def operation_logic():
        df, file_id = read_csv_from_drive(service)
        if df is None:
            return False
        original_len = len(df)
        try:
            datetime.datetime.strptime(period, '%Y')
            df = df[~df['time'].str.startswith(period)]
        except ValueError:
            try:
                datetime.datetime.strptime(period, '%Y-%m')
                df = df[~df['time'].str.startswith(period)]
            except ValueError:
                try:
                    datetime.datetime.strptime(period, '%Y-%m-%d')
                    df = df[~df['time'].str.startswith(period)]
                except ValueError:
                    console.print(f"[red]Formato data non valido. Usa 'yyyy', 'yyyy-mm' o  
↪ 'yyyy-mm-dd'[/red]")
                    return False
        if len(df) == original_len:
            return False
        else:
            upload_csv_to_drive(service, df, file_id)
            return True

    success = progress_bar(operation_logic)
    if success:
        console.print(f"\n[green]Righe con il periodo {period} eliminate con  
↪ successo[/green]")
    else:
        console.print(f"\n[yellow]Nessuna riga trovata con il periodo {period}[/yellow]")

def clear_file(service):
    def operation_logic():
        df, file_id = read_csv_from_drive(service)
        if df is None:
            return False
        df = pd.DataFrame(columns=HEADERS)
        upload_csv_to_drive(service, df, file_id)
        return True
```

```
success = progress_bar(operation_logic)
if success:
    console.print(f"\n[green]File '{FILENAME}' svuotato con successo.[/green]")
```

When the user chooses to display the operations, the `show_file` function retrieves the data from the CSV and displays it using a well-formatted table. The user can filter the data by year, month or day, using the format `yyyy`, `yyyy-mm` or `yyyy-mm-dd`.

```
def show_file(service, period=None):
    try:
        df, _ = read_csv_from_drive(service)
        if df is None:
            return

        if period:
            try:
                datetime.datetime.strptime(period, '%Y')
                filtered_df = df[df['time'].str.startswith(period)]
            except ValueError:
                try:
                    datetime.datetime.strptime(period, '%Y-%m')
                    filtered_df = df[df['time'].str.startswith(period)]
                except ValueError:
                    try:
                        datetime.datetime.strptime(period, '%Y-%m-%d')
                        filtered_df = df[df['time'].str.startswith(period)]
                    except ValueError:
                        console.print("[red]Formato data non valido. Usa 'yyyy', 'yyyy-mm' o  
↪ 'yyyy-mm-dd'.[/red]")
                        return

        df = filtered_df

        table = Table(title="\nIoT Operations", show_lines=True)
        table.add_column("Time", justify="center", style="cyan")
        table.add_column("Operation", justify="center", style="magenta")

        if df.empty:
            console.print("[yellow]La tabella è vuota.[/yellow]")
        else:
            for _, row in df.iterrows():
                table.add_row(row['time'], row['operation'])

        console.print(table)
    except Exception as e:
        console.print(f"[red]Errore durante la visualizzazione del file: {e}[/red]")
```

### 2.2.2 Insecure Network

Creating a network for a cloud-based IoT architecture requires careful planning and the implementation of key components to ensure efficiency, security and ease of management. At the heart of this infrastructure is the Raspberry Pi, which serves as the central hub for connecting and communicating between IoT devices and the cloud infrastructure.

### Initial Setup

The first step is to install the necessary software on the Raspberry Pi. Of these, *hostapd* and *dnsmasq* play key roles. *Hostapd* is responsible for configuring the Raspberry Pi as a wireless access point, allowing IoT devices to connect to the local network. *Dnsmasq*, on the other hand, acts as a DHCP server, automatically assigning IP addresses to devices connecting to the network, thus simplifying address management and reducing the need for manual configuration.

```
sudo apt install -y hostapd dnsmasq

sudo systemctl unmask hostapd
sudo systemctl enable hostapd
```

Then I set the WLAN language to IT with:

```
sudo raspi-config nonint do_wifi_country IT
```

### Configuring the Static IP Address

To ensure network stability and predictability, it is essential to assign a static IP address to the Raspberry Pi. This prevents the device's IP address from changing after a reboot or following other changes to the network, ensuring that IoT devices can always find and communicate with the Raspberry Pi without interruption. Furthermore, as IoT devices do not need to communicate with the outside world but only with each other, IP forwarding is disabled, thus reducing the risk of unauthorized access and external attacks.

```
interface wlan0
    static ip_address=192.168.14.240/24
    nohook wpa_supplicant
```

### Setup dnsmasq

Thanks to dnsmasq, the Raspberry Pi also acts as a DHCP server, assigning a pool of IP addresses to new devices that connect, facilitating network expansion without unexpected errors. I modified the `dnsmasq.conf` file by adding these lines to the end of the file:

```
interface=wlan0
dhcp-range=192.168.14.20,192.168.14.30,255.255.255.0,24h
```

I give a higher range so that there are no problems; the lease is set for 24 hours after which the device is assigned a new IP.

### Setup Access Point

The Raspberry Pi also assumes the role of an access point by managing the wireless connections of IoT devices and creating a dedicated local network. To do this, I modified the `hostapd` configuration file:

```
country_code=IT
interface=wlan0
driver=nl80211

hw_mode=g
ieee80211n=1
channel=7
wmm_enabled=0
macaddr_acl=0
ignore_broadcast_ssid=0

auth_algs=1
wpa=2
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP

ssid=Raspberry-AP
wpa_passphrase=Password123
```

In the configuration fragment presented, the country code is specified as Italy (`country_code=IT`) to comply with local radio frequency regulations. The wireless interface used is `wlan0` with the `nl80211` driver, which is suitable for modern Linux network cards. The network operates in the 2.4 GHz band (`hw_mode=g`) with the IEEE 802.11g standard, and 802.11n is enabled to improve performance and range. The chosen channel is 7, and Wireless Multimedia Extensions (WMM) are disabled to simplify configuration.

For security, WPA2 (`wpa=2`) is enabled with key management via a pre-shared key (`wpa_key_mgmt=WPA-PSK`), using TKIP and CCMP encryption protocols to protect transmitted data. MAC address checking is disabled (`macaddr_acl=0`), allowing any device to connect, while the network SSID (`Raspberry-AP`) is made visible (`ignore_broadcast_ssid=0`). Finally, a passphrase (`Password123`) is set to act as an access key to ensure that only authorized users can connect to the network. Then, I modify the `/etc/default/hostapd` file adding the line:

```
DAEMON_CONF='/etc/hostapd/hostapd.conf'
```

In the end, I reboot the Raspberry Pi.

### Setup Broker Mosquitto

Another feature that the Raspberry Pi hosts is the Mosquitto broker, an essential component for communication between IoT devices and the cloud infrastructure. Mosquitto uses the MQTT protocol to efficiently and lightly manage message transmission, enabling reliable and scalable communication between devices. First, I install the Mosquitto broker:

```
sudo apt install mosquitto mosquitto-clients -y
```

Then, I change the Mosquitto configuration file in this way:

```
listener 1883
protocol mqtt
allow_anonymous true

# log settings
log_type error
log_type warning
log_type notice
log_type information
```

This configuration block sets Mosquitto to listen for MQTT connections on the standard port 1883, allows anonymous connections to facilitate IoT device access, and defines the level of log detail for effective broker management and monitoring. This is an insecure configuration because it has no authentication mechanism; in fact, an attacker can gain unauthorized access, intercept and manipulate the flow of data (MITM), carry out DoS attacks and exhaust system resources.

Next, I will show a secure solution by appropriately configuring the access point and the Mosquitto broker.

### 2.2.3 Secure Network

To ensure the security of the IoT architecture, it is essential to make significant changes to both the access point configured via hostapd and the Mosquitto broker. Regarding the access point configuration, it is advisable to limit the number of devices that can connect by setting a maximum number of users and allowing connections only from specific MAC addresses. This approach ensures that only authorized devices can access the network, significantly reducing the risk of unauthorized access. On the Mosquitto broker front, it is crucial to disable anonymous connections and configure an authentication system based on username and password, ensuring that only authorized devices can connect to the broker. Furthermore, implementing TLS/SSL to encrypt MQTT traffic protects data in transit from eavesdropping and tampering, ensuring the integrity and confidentiality

of communications. Strict topic access rules should also be defined, limiting publication and subscription operations to only those topics necessary for each device, thus reducing the attack surface. Finally, keeping both the access point and Mosquitto broker software up-to-date is crucial to protect against known vulnerabilities and emerging exploits. By implementing these changes, the IoT architecture remains secure and resilient against intrusion attempts and cyberattacks.

### Setup secure Access Point

Updates made in the hostapd configuration file include MAC address-based filters, limiting the maximum number of connected devices and updating the network passphrase.

```
max_num_sta=3
macaddr_acl=1
accept_mac_file=/etc/hostapd/hostapd.accept

ssid=Raspberry-AP
wpa_passphrase=%3LaB6_!oT4%
```

The `max_num_sta=3` parameter imposes a maximum limit of three devices simultaneously connected to the access point. This helps prevent network overloads and reduces the risk of unauthorized access by unwanted devices. With `macaddr_acl=1` I have filtering based on MAC addresses, allowing only those devices specified in the accept file to connect to the network. This additional security mechanism ensures that only pre-approved devices can access the network, significantly reducing the risk of unauthorized access. The allowed MAC file was configured like this:

```
# Esp8266
DC:4F:22:60:82:BF

# Esp32
78:21:84:89:02:E4

# Arduino
34:94:54:23:13:58
```

### Setup secure broker Mosquitto

The changes made to the Mosquitto broker configuration file are intended to ensure that only authorized devices can access the broker and that all communication takes place in a secure and encrypted manner.

First, I defined the default user as root, ensuring that critical operations are performed with appropriate privileges:



## 2. MY CLOUD-BASED IOT ARCHITECTURE

```
user root
```

I configured Mosquitto by clearly specifying the file paths to the certificates needed to establish secure connections. These certificates were generated with OpenSSL for each device, ensuring that each connection is authenticated and encrypted correctly.

```
cafile /etc/mosquitto/certs/ca.crt
certfile /etc/mosquitto/certs/broker.crt
keyfile /etc/mosquitto/certs/broker.key
require_certificate true
```

In addition, I disabled anonymous connections by setting `allow_anonymous false`, thus requiring authentication via username and password for each device. Credentials are managed via the password file, while access permissions to the various topics are defined in the acl file. I have specified that only authenticated users can access the topics, clearly defining the operations allowed for each user.

```
allow_anonymous false
password_file /etc/mosquitto/passwd
acl_file /etc/mosquitto/aclfile
```

In the ACL file, I specifically defined permissions for each user. The Raspberry Pi can receive and send messages to the outside and to the ESP8266. The ESP8266 can send and receive messages from all three IoT devices, while the ESP32 and the Arduino can only receive and send messages from the ESP8266 device. Below we can see the code and in the Figure 2.2 the logical path of information.

```
user broker
topic readwrite raspberry/topic

user esp8266
topic write raspberry/topic
topic write esp32/topic
topic write arduino/topic
topic read esp8266/topic

user esp32
topic write esp8266/topic
topic read esp32/topoc

user arduino
topic write esp8266/topic
topic read arduino/topic
```

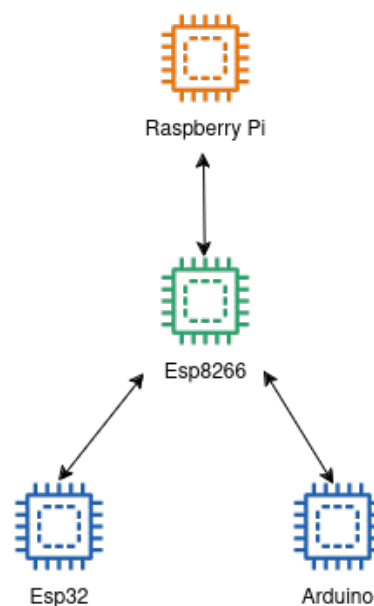


Figure 2.2: Communication logic.

The TLS settings were strengthened by specifying the minimum version of TLS to be used, namely `tlsv1.2`. I did not enter 1.3 as the minimum version since it is not supported by Esp8266 and Arduino.

```
tls_version tlsv1.2
```

### Setup NTP server

To ensure the correct use of SSL/TLS certificates in my IoT architecture, it was necessary to install and configure a Network Time Protocol (NTP) server on the Raspberry Pi. Digital certificates rely on precise timestamps to validate their issue and expiry dates, and accurate synchronization of the system clock is crucial to avoid certificate validity issues. Without a synchronized clock, IoT devices could detect certificates as invalid if the Raspberry Pi's time does not match the current time, thus compromising secure communications between devices and the Mosquitto broker.

To implement this solution, the `ntp` service was installed on the Raspberry Pi, configuring the configuration file `/etc/ntp.conf` to use reliable public NTP servers. Once configured, the NTP service starts automatically at system start-up, keeping the Raspberry Pi's time constantly updated. In parallel, the IoT boards were configured to synchronise their system clocks with the Raspberry Pi via sketch with the Arduino IDE.

## 2.3 Configuration of IoT Devices

As part of the configuration of the Cloud-based IoT architecture, the other devices in the network, such as the Arduino, ESP8266 and ESP32, were configured using the Arduino IDE by implementing specific sketches. This approach made it possible to programme and customize the behaviour of each device in an efficient and flexible manner. In order to evaluate the impact of the implemented security measures, two distinct versions of the codes were created: a secure configuration and an insecure configuration.

The insecure configuration represents a basic version of the sketches, lacking authentication and encryption mechanisms. In this version, devices connect to the Mosquitto broker without verifying the identity of the server or other clients, and transmitted data is not protected by encryption. This configuration serves as a starting point for understanding the inherent vulnerabilities of an unprotected IoT network, highlighting the risks associated with unauthorized access and the possibility of data interception and manipulation. In contrast, the secure configuration incorporates several advanced security measures. Each device has been programmed to use SSL/TLS certificates previously generated with OpenSSL, thus guaranteeing mutual authentication between the devices and the broker. In addition, data exchanged via the MQTT protocol is encrypted, preventing third parties from intercepting or altering communications. This configuration also includes credential

management via a secure password file and the implementation of access controls based on the ACLs defined in the Mosquitto broker. These measures ensure that only authorized devices can participate in the network and that transmitted information is protected from potential external attacks.

Through the implementation of both configurations, it was possible to compare and analyse the effectiveness of the security measures taken, demonstrating how a correct configuration can significantly increase the robustness and reliability of the entire IoT infrastructure.

## Chapter 3

# Creation of a Real IoT Attack Dataset

In this chapter, the process of building a specific dataset to analyses attacks aimed at a cloud-based IoT architecture will be illustrated. In particular, the types of attacks carried out will be detailed. The methodologies adopted to record the network traffic generated during the attacks and how this raw data was subsequently processed and converted for use in Machine Learning models will be discussed. The extracted features and their relevance in the context of the analysis will be described and analysed. In addition, the organization of the final dataset will be described, including data labeling and partitioning strategies. Finally, an initial critical comparison will be made with the TON\_IoT dataset to highlight similarities and significant differences. This comparison will serve as a basis for subsequent experiments and to better understand the effectiveness of the developed dataset in detecting and preventing attacks in the cloud-based IoT environment.

### 3.1 Types of Attacks Performed

In developing my dataset, I meticulously selected a variety of cyber-attacks to simulate realistic threats against the cloud-based IoT architecture. These attacks are divided into four main classes: Denial of Service (DoS), Brute Force Attacks, MQTT Protocol Attacks, and Reconnaissance (Recon) Attacks. The choice of these specific attacks was motivated by their prevalence and generic nature in the cybersecurity landscape, making them representative of common threats faced by IoT systems. Including multiple attacks within each class was a deliberate strategy to ensure that the dataset comprehensively represents these threats. By selecting attacks prevalent in real-world scenarios for each class, I aimed to provide a robust foundation for training machine learning models to detect and mitigate these threats effectively.

Therefore, the overarching motivation behind these choices is to create a realistic and practical resource that aligns with common security challenges faced in IoT environments, thereby contributing to the advancement of effective security solutions.

#### 3.1.1 Denial of Service Attacks

In the DoS class, I employed a variety of flooding and fragmentation attacks using *hping3* [39]. Attacks such as SYN Flood, ICMP Flood, UDP Flood, and TCP Flood are among the most widespread methods used by attackers to overwhelm network resources. By including fragmentation attacks like ACK Fragmentation and ICMP Fragmentation, I address techniques where attackers attempt to bypass security defenses by sending fragmented packets that can disrupt packet reassembly processes. The diversity of these attacks covers both volume-based and protocol-specific DoS methods, ensuring that the dataset encapsulates the spectrum of DoS threats commonly encountered.

The DoS attacks effectively exhausted the resources of the Raspberry Pi devices. The relentless flood of traffic overwhelmed the CPU and memory, causing the Raspberry Pi to become unresponsive and leading to system crash. This illustrates how resource-constrained IoT devices are particularly susceptible to DoS attacks, which can easily incapacitate them by saturating their limited processing capabilities.

For each type of dos attack, three different versions were launched (*fast*, *faster*, *flood*) which differ in the rate at which the packets are sent. Below are the commands used and the statistics calculated:

```
# ACK Fragmentation
sudo hping3 -A -f --fast [TARGET_IP]      Sent: 720 - Lost: 1% - Received: 719
sudo hping3 -A -f --faster [TARGET_IP]    Sent: 315487 - Lost: 100% - Received: 0
sudo hping3 -A -f --flood [TARGET_IP]     Sent: 304309 - Lost: 100% - Received: 0

# ICMP Flood
sudo hping3 --icmp --fast [TARGET_IP]     Sent: 624 - Lost: 1% - Received: 623
sudo hping3 --icmp --faster [TARGET_IP]   Sent: 706676 - Lost: 100% - Received: 0
sudo hping3 --icmp --flood [TARGET_IP]    Sent: 696309 - Lost: 100% - Received: 0

# RSTFIN Flood
sudo hping3 -R -F --fast [TARGET_IP]      Sent: 617 - Lost: 100% - Received: 0
sudo hping3 -R -F --faster [TARGET_IP]    Sent: 921374 - Lost: 100% - Received: 0
sudo hping3 -R -F --flood [TARGET_IP]     Sent: 995106 - Lost: 100% - Received: 0

# PSHACK Flood
sudo hping3 -P -A --fast [TARGET_IP]      Sent: 622 - Lost: 1% - Received: 621
sudo hping3 -P -A --faster [TARGET_IP]    Sent: 960114 - Lost: 100% - Received: 0
sudo hping3 -P -A --flood [TARGET_IP]     Sent: 1026014 - Lost: 100% - Received: 0

# ICMP Fragmentation
sudo hping3 --icmp -f --fast [TARGET_IP]   Sent: 602 - Lost: 0% - Received: 602
sudo hping3 --icmp -f --faster [TARGET_IP] Sent: 724654 - Lost: 100% - Received: 0
sudo hping3 --icmp -f --flood [TARGET_IP]  Sent: 667512 - Lost: 100% - Received: 0

# TCP Flood
sudo hping3 --fast [TARGET_IP]            Sent: 613 - Lost: 1% - Received: 612
sudo hping3 --faster [TARGET_IP]          Sent: 834741 - Lost: 100% - Received: 0
sudo hping3 --flood [TARGET_IP]           Sent: 913504 - Lost: 100% - Received: 0
```

---

### 3. CREATION OF A REAL IOT ATTACK DATASET

---

```
# SYN Flood
sudo hping3 -S --fast [TARGET_IP]    Sent: 607 - Lost: 1% - Received: 606
sudo hping3 -S --faster [TARGET_IP]  Sent: 948481 - Lost: 100% - Received: 0
sudo hping3 -S --flood [TARGET_IP]   Sent: 1026853 - Lost: 100% - Received: 0

# SynonymousIP Flood
sudo hping3 -S --rand-source --fast [TARGET_IP]    Sent: 724 - Lost: 100% - Received: 0
sudo hping3 -S --rand-source --faster [TARGET_IP]  Sent: 276474 - Lost: 100% - Received: 0
sudo hping3 -S --rand-source --flood [TARGET_IP]   Sent: 264559 - Lost: 100% - Received: 0

# UDP Flood
sudo hping3 --udp --fast [TARGET_IP]    Sent: 628 - Lost: 90% - Received: 68
sudo hping3 --udp --faster [TARGET_IP]  Sent: 820985 - Lost: 100% - Received: 0
sudo hping3 --udp --flood [TARGET_IP]   Sent: 873689 - Lost: 100% - Received: 0

# UDP Fragmentation
sudo hping3 --udp -f --fast [TARGET_IP]    Sent: 606 - Lost: 90% - Received: 66
sudo hping3 --udp -f --faster [TARGET_IP]  Sent: 730130 - Lost: 100% - Received: 0
sudo hping3 --udp -f --flood [TARGET_IP]   Sent: 876046 - Lost: 100% - Received: 0
```

#### 3.1.2 Brute Force Attack

I chose brute force attack on the SSH protocol using *Hydra* [40] because SSH is a ubiquitous service for secure remote access in IoT devices and networks. Brute force attacks are a fundamental and widespread method used by attackers to gain unauthorized access by systematically trying combinations of usernames and passwords. Including this attack highlights the importance of strong authentication mechanisms and allows the dataset to capture data related to authentication attempts and failures, which are critical for intrusion detection systems.

The brute force attack on the SSH protocol was executed using a wordlist of common passwords. Due to the simplicity of the password used on the target device, the attack quickly succeeded in gaining unauthorized access. Below there is the commands used:

```
hydra -s 22 -v -V -L /usr/share/wordlist/rockyou.txt -x 1:1:a -e s -t 8
↪ 192.168.14.240 ssh
```

#### 3.1.3 MQTT Protocol Attacks

The MQTT protocol is a cornerstone in IoT communications due to its lightweight nature. Attacks like Flooding DoS using *mqtt-malaria* [41], and Malformed Data and SlowITe attacks using *mqttsa* [42], exploit common vulnerabilities in MQTT implementations. Flooding the MQTT broker (Raspberry Pi) with excessive messages can lead to denial of service, while sending malformed data can crash or disrupt the broker's operation. SlowITe attacks involve sending data at a slow rate to consume resources gradually. By incorporating these attacks, I aim to cover typical MQTT-related threats, thereby enhancing the dataset's relevance for securing IoT communication protocols.

---

### 3. CREATION OF A REAL IOT ATTACK DATASET

---

The MQTT protocol attacks resulted in significant disruptions. The Flooding DoS attack overwhelmed the Raspberry Pi with an excessive number of messages, leading to denial of service for legitimate clients. The Malformed Data and SlowITe attacks exploited weaknesses in the MQTT implementation, causing erratic behavior and crashes in the broker and connected devices.

#### Flooding DoS

In the Flooding DoS, each command differs in the `-T` parameter, which controls the message throughput (the rate at which messages are sent per second). The first command sets the rate at 500 messages per second, the second increases it to 1,000 messages per second and the third escalates it further to 5,000 messages per second. ByBy progressively increasing the message rate, each command intensifies the load on the broker, leading to service degradation or complete failure.

```
# Flooding DoS
python2 malaria publish -P 10 --host 192.168.14.240 --msg_count 100000 -T 500
python2 malaria publish -P 10 --host 192.168.14.240 --msg_count 100000 -T 1000
python2 malaria publish -P 10 --host 192.168.14.240 --msg_count 100000 -T 5000
```

#### SlowITe Attack

In contrast, the SlowITe attacks leverage the `mqtttsa.py` script to exploit the MQTT broker's resource management by initiating a large number of simultaneous connections that send data very slowly. The key difference between the two SlowITe commands lies in the `-fcsz` parameter, which specifies the *frame chunk* size (the size of data chunks sent in each frame). The first command uses a frame chunk size of 10 bytes, while the second uses 25 bytes. By adjusting the frame chunk size, the attacker manipulates how slowly data is transmitted over each of the 12,000 simulated client connections, as indicated by the `-sc 12000` parameter. Smaller frame sizes result in slower data transmission, increasing the duration of each connection and consuming more of the broker's resources over time. This method differs from the Flooding DoS attacks in that it doesn't rely on high volumes of traffic but instead exhausts the broker's resources by maintaining numerous slow connections, ultimately leading to a denial of service due to resource exhaustion.

```
# SlowITe
python mqtttsa.py -v 2 -fcsz 10 -sc 12000 192.168.14.240
python mqtttsa.py -v 2 -fcsz 25 -sc 12000 192.168.14.240
```

#### Malformed Data Attack

Then, I executed a Malformed Data attack against the Raspberry Pi using the command:

```
# Malformed Data
python3 mqtttsa.py --md 192.168.14.240
```

This command utilizes the `mqtttsa.py` script, which is designed to perform security assessments on MQTT brokers by simulating various attack scenarios. The `-md` flag specifically triggers the Malformed Data attack mode within the script.

By running this command, I directed the script to connect to the MQTT broker located at the IP address 192.168.14.240 and send improperly structured or corrupted MQTT packets. The purpose of this attack is to test the broker's ability to handle unexpected or non-compliant data formats, which can reveal vulnerabilities in the broker's input validation and error-handling mechanisms.

#### 3.1.4 Reconnaissance Attacks

Reconnaissance attacks are crucial for attackers to gather information about the target network before launching more destructive attacks. I used tools like *Nmap* [43] to perform host discovery, OS detection and port scanning, while *Nessus* [44] and *Vulscan* [43] to perform vulnerability scanning. By employing various scanning techniques on devices such as Arduino, ESP32, ESP8266, and Raspberry Pi, I simulate a comprehensive reconnaissance process.

#### Vulnerability Scanning

For vulnerability scanning, I used *Vulscan* launched the following script:

```
sudo nmap -sV -p 1-65535 --open --script vulscan/vulscan.nse --script-args
↳ vulscandb=all 192.168.14.0/24
```

The command initiates a comprehensive scan of the entire IP range in the subnet 192.168.14.0/24, which includes all addresses from 192.168.14.1 to 192.168.14.254. The `-sV` option enables version detection, prompting Nmap to probe open ports to determine the service and version information running on them. This goes beyond simply identifying that a port is open; it gathers detailed information about the services behind those ports, which is crucial for vulnerability assessment.

By specifying `-p 1-65535`, the command instructs Nmap to scan all possible TCP ports, from port 1 to port 65,535. This exhaustive port range ensures that no potential service is missed, including those operating on non-standard or high-numbered ports. The `-open` parameter tells Nmap to display only open ports in the output, filtering out closed or filtered ports and focusing on services that are actively accepting connections.



---

### 3. CREATION OF A REAL IOT ATTACK DATASET

---

The inclusion of `-script vulscan/vulscan.nse` incorporates the Vulscan script into the scanning process. Vulscan is an extension that enhances Nmap's capabilities by performing vulnerability scanning. It takes the service and version information identified by the `-sV` option and cross-references it against multiple vulnerability databases to identify known security issues associated with those services.

The argument `-script-args vulscandb=all` directs Vulscan to use all available vulnerability databases for its checks. This makes the vulnerability assessment more comprehensive, as it references a wide array of sources, including Common Vulnerabilities and Exposures (CVE) listings, security advisories, and exploit repositories.

#### OS discovery

For the OS discovery I used the following command:

```
sudo nmap -O 192.168.14.xxx
```

Where xxx indicates the last octet of each associated device.

#### Host Discovery

For the host discovery the commands that I used are:

```
# Disable port scanning. Host discovery only.
sudo nmap -sn 192.168.14.0/24

# Disable host discovery. Port scan only.
sudo nmap -Pn 192.168.14.0/24

# TCP SYN discovery on port x.Port 80 by default
sudo nmap -PS 192.168.14.0/24

# TCP ACK discovery on port x.Port 80 by default
sudo nmap -PA 192.168.14.0/24

# UDP discovery on port x.Port 40125 by default
sudo nmap -PU 192.168.14.0/24

# ICMP Echo Scan (Ping scan)
sudo nmap -PE 192.168.14.0/24

# ICMP Timestamp Scan
sudo nmap -PP 192.168.14.0/24

#ICMP Address Mask Scan
sudo nmap -PM 192.168.14.0/24
```

```
# ARP discovery on local network
sudo nmap -PR 192.168.14.0/24
```

#### Port Discovery

In the end, the port scan was performed using the following commands:

```
# TCP Syn port scan
sudo nmap -sS -p- <target>

# TCP connect port scan (Default without root privilege)
nmap -sT -p- <target>

# TCP Null scan
sudo nmap -sN -p- <target>

# TCP FIN scan
sudo nmap -sF -p- <target>

# TCP Xmas scan
sudo nmap -sX -p- <target>

# TCP ACK port scan
sudo nmap -sA -p- <target>

# TCP Window port scan
sudo nmap -sW -p- <target>

# TCP Maimon port scan
sudo nmap -sM -p- <target>

# UDP Scan
sudo unicornscan -m U -Iv <target>:1-65535
```

Where <target> indicates the ip of an IoT device.

## 3.2 Logging Network Traffic

The logging of network traffic was the main operation throughout the thesis work. This process involves implementing a specific architecture for capturing and analysing traffic, as well as defining detailed scanning patterns to distinguish between legitimate traffic and potential threats. The following is a detailed description of the steps taken to ensure effective and accurate monitoring of network traffic.

### 3.2.1 Architecture for Traffic Sniffing

The architecture adopted for logging network traffic consists of several interconnected components that work together to capture and analyse the data transmitted within the IoT network. At the core of this architecture is an attacker PC and a passive sniffing PC, both connected to the Raspberry Pi via a WLAN connection, with the Raspberry Pi acting as a hotspot. This configuration ensures that both PCs are positioned in the same subnet, facilitating direct interaction with the cloud-based IoT infrastructure. The attacker PC is responsible for initiating targeted attacks against the IoT architecture, generating traffic that needs to be monitored and logged for later analysis. In parallel, the passive sniffing PC is dedicated to capturing network traffic in passive mode, thus ensuring a comprehensive view of the interactions within the network (Figure 3.1).

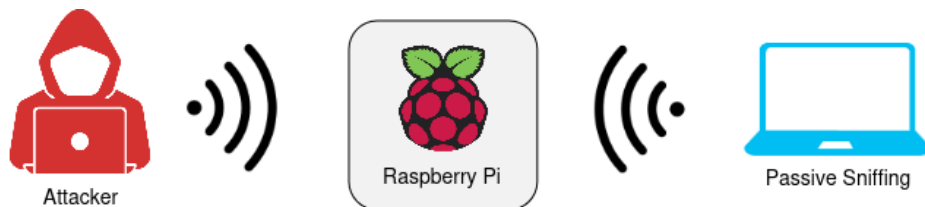


Figure 3.1: Traffic Sniffing Architecture.

The passive sniffing PC uses specific tools such as *airmon-ng* and *airodump-ng* to put the wireless interface in monitor mode and start sniffing traffic. The commands used to initiate this procedure are:

```
airmon-ng start wlan0
```

To start collecting network packets:

```
airodump-ng wlan0mon -c 7 --bssid B8:27:EB:D2:11:53 -w Raspberry-AP
```

Once the sniffing process has started, *Wireshark* is used to capture and analyze data packets in detail.

### 3.2.2 The Dataset in detailed

#### 3.2.2.1 Scan times

Network traffic scans were structured according to predefined patterns that alternate between periods of normal traffic and periods of attack, in order to simulate realistic and variable scenarios that could occur within the IoT infrastructure. These patterns allow a clear distinction between legitimate activities and potential threats, facilitating analysis and management of system security.

**Normal traffic** is divided into two main categories: “*idle*” traffic and “*normal*” traffic. “Idle” traffic refers to periods of inactivity when users do not interact with the system, for a total of five hours. In contrast, “normal” traffic represents times when users actively use the system, also for a duration of five hours. This breakdown provides a clear baseline for distinguishing between moments of actual use and periods of inactivity.

As for **attack scans**, each scan starts with two minutes of legitimate traffic, followed by the attack, and ends with an additional two minutes of legitimate traffic. The duration of the attack varies depending on the type of threat. In the case of DoS attacks, the attack phase has a duration of one minute, after which it is stopped, allowing the system to return to a normal traffic state. In contrast, for MQTT attacks, the duration of the attack is two minutes, before it is stopped. This structure allows us to assess the impact of the attacks on the system and its resilience during and after the attempted compromise.

The Table 3.1 provides a summary of the times calculated for scans, taking macro attacks into consideration rather than going into the details of each specific attack. Each attack category groups its internal variants, since the times are identical for all micro-attacks within each category. In addition, the table also shows the duration of only benign traffic. The section of the table marked with / indicates a variable duration where traffic does not follow a fixed pattern, while *none* indicates a section not affected by the specified times. The table summarizes the legitimate traffic, attack period, and end times for each type of scan performed.

Scan Type	Initial Traffic	Attack Time	Final Traffic
Benign Traffic	5 hrs	none	none
Brute Force Attack	2 mim	/	2 min
Recon Attack	2 mim	/	2 min
DoS Attack	2 mim	1 min	2 min
MQTT Attack	2 min	2 min	2 min

Table 3.1: Structure of the Scans and Attack Durations

#### 3.2.2.2 Splitting the dataset

Following the execution of the scans, two datasets were created:

1. without MQTT attacks (4 classes - Dos, Brute Force, Benign, Recon);
2. includes MQTT attacks (5 classes - Dos, Brute Force, Benign, Recon, MQTT Attack).

The dataset generated without considering MQTT attacks contains a total of **51,838,560** samples, divided into the following classes (Table 3.2):

Class	# Samples	Percentage (%)
dos	30,654,284	59.13
brute_force	34,601	0.07
benign	91,607	0.18
recon	21,058,068	40.62
<b>Total</b>	<b>51,838,560</b>	<b>100.00</b>

Table 3.2: Number of Samples for each Class (*without MQTT attacks*).

The dataset that includes MQTT attacks has a total of **51,917,217** samples, with the following class distribution (Table 3.3):

Class	# Samples	Percentage (%)
dos	30,654,284	59.04
brute_force	34,601	0.07
benign	93,517	0.18
recon	21,058,068	40.56
mqtt_attack	76,747	0.15
<b>Total</b>	<b>51,917,217</b>	<b>100.00</b>

Table 3.3: Number of Samples for each Class (*with MQTT attacks*).

#### 3.2.2.3 Features Selection

The features chosen are **32** and they were carefully selected to represent important aspects of network traffic, both for anomaly detection and for identifying specific types of attacks. Below (Table 3.4) is an overview of the included features and their description:

### 3. CREATION OF A REAL IOT ATTACK DATASET

Feature in Tshark	Feature in Dataset	Description
type.attack	Attack_Type	Type of attack or benign traffic
rate	Rate	Rate of packet transmission
ip.ttl	Time_To_Leave	Number of hops allowed before packet is discarded
ip.hdr.len	Header_Length	Length of the IP header
_ws.col.Protocol	Protocol_Type	Type of protocol used
tcp.flags.fin	TCP_Flag_FIN	FIN flag for TCP connection termination
tcp.flags.syn	TCP_Flag_SYN	SYN flag for TCP connection initiation
tcp.flags.reset	TCP_Flag_RST	RST flag indicating connection reset
tcp.flags.push	TCP_Flag_PSH	PSH flag for pushing data to the receiver
tcp.flags.ack	TCP_Flag_ACK	ACK flag acknowledging packet receipt
tcp.flags.ece	TCP_Flag_ECE	ECE flag for congestion notification
tcp.flags.cwr	TCP_Flag_CWR	CWR flag for congestion window reduced
frame.len	Packet_Length	Length of the entire packet
frame.time_delta	IAT	Inter-arrival time between packets
ip.flags.mf	Packet_Fragments	Indicates whether the packet is fragmented
tcp.len	TCP_Length	Length of TCP payload
mqtt.conack.flags	MQTT_ConAck_Flags	Flags for MQTT connection acknowledgment
mqtt.conflag.cleansess	MQTT_CleanSession	Clean session flag for MQTT
mqtt.qos	MQTT_QoS	Quality of Service level for MQTT messages
mqtt.conflag.reserved	MQTT_Reserved	Reserved flag in MQTT connection
mqtt.retain	MQTT_Retain	Retain flag for MQTT messages
mqtt.conflag.willflag	MQTT_WillFlag	Will flag indicating last will in MQTT
mqtt.conflags	MQTT_ConFlags	Connection flags for MQTT
mqtt.dupflag	MQTT_DupFlag	Duplicate flag for retransmitted MQTT messages
mqtt.hdrflags	MQTT_HeaderFlags	Header flags in MQTT messages
mqtt.kalive	MQTT_KeepAlive	Keep-alive interval for MQTT connections
mqtt.len	MQTT_Length	Length of MQTT payload
mqtt.msgtype	MQTT_MessageType	Type of MQTT message
mqtt.proto.len	MQTT_Proto_Length	Length of MQTT protocol name
mqtt.conflag.qos	MQTT_Conflag_QoS	Quality of Service flag for MQTT connection
mqtt.conflag.retain	MQTT_Conflag_Retain	Retain flag for MQTT connection
mqtt.ver	MQTT_Version	Version of the MQTT protocol

Table 3.4: Features Selection.

Selected features, such as TCP flags and Inter-Arrival Time (IAT) of packets, are essential for identifying abnormal behavior in network traffic. IAT, for example, allows monitoring the frequency with which packets are received. Excessively short inter-arrival times may suggest the presence of Denial of Service (DoS) attacks, in which a large number of packets are sent in rapid succession to overwhelm a device or network.

Similarly, TCP flags, such as SYN, ACK, FIN and RST indicate the state of a TCP connection and can be used to detect anomalies in connection behavior. For example, a high frequency of packets with the SYN flag may indicate an attempted SYN Flood attack, in which the attacker attempts to exhaust a server's resources by flooding it with connection requests without completing them. Such anomalous patterns are clear indicators of malicious behavior, which an IDS must be able to recognize to activate appropriate countermeasures.

The MQTT protocol, which is designed for lightweight and reliable communications between IoT devices, is vulnerable to various types of attacks, including MQTT Flood and Session Hijacking. Features such as Clean Session, Quality of Service (QoS), and Retain Flag allow monitoring of how MQTT connections are handled. Abnormalities in these variables, such as unanticipated changes in the QoS level or the presence of unexpected flags, can indicate an attempt to manipulate the connection, hijack the session, or flood the network with MQTT messages. Monitoring these variables is therefore essential to ensure the security of IoT devices and the continuity of their communication.

Selected features also enable a clear distinction between legitimate and malicious traffic. This distinction is particularly important to ensure that the IDS system can minimize false positives, i.e., errors in which benign traffic is classified as malicious.

#### 3.2.2.4 PCAP to CSV

To create the dataset, pcapng format files were converted to CSV files using the *tshark* tool. This process was automated using a Python script, which allowed large volumes of data to be handled efficiently and accurately.

Initial conversion of pcapng files to CSV with tshark allowed all important features to be extracted. Once the CSV files were obtained, it was necessary to unify the numerical data by converting all numerical variables to float64 format. This type of conversion is critical to ensure the accuracy and consistency of the data when applying Machine Learning algorithms. During this step, special care was taken to avoid conversion errors, such as handling missing or inconsistent values that could have compromised the quality of the dataset.

Once the conversion was completed, the various CSV files were combined to create one large dataset that included all types of traffic: both benign and malicious traffic, represented by the various attacks mentioned earlier. However, to ensure the quality of the

dataset and the effectiveness of the subsequent training process, a number of filters and transformations had to be applied. The main goal was to create a balanced dataset in which the number of samples was uniform for each class. To do this, all rows containing less-used protocols were removed, keeping only those associated with more than 1,000 samples, as they were more common and relevant to the analysis. This selection reduced complexity and ensured that only the most representative classes of protocols were included in the final dataset.

Once the least relevant protocols were filtered out, it was necessary to balance the dataset in terms of the number of samples for each class. To this end, the class with the lowest number of samples was identified, and then, for each class in the dataset, a number of samples equal to the number of samples in the minority class were randomly selected. This approach resulted in a balanced dataset (with **34,601 samples** for each class), useful for training Machine Learning models without the risk of introducing bias toward the most represented classes.

### 3.3 Comparison between TON\_IoT and my Dataset

The TON\_IoT datasets are new generations of Internet of Things (IoT) and Industrial IoT (IIoT) datasets for evaluating the fidelity and efficiency of different cybersecurity applications based on Artificial Intelligence. The datasets have been called ‘ToN\_IoT’ as they include heterogeneous data sources collected from Telemetry datasets of IoT and IIoT sensors, Operating systems datasets of Windows 7 and 10 as well as Ubuntu 14 and 18 TLS and Network traffic datasets. The datasets were collected from a realistic and large-scale network designed at the IoT Lab of the UNSW Canberra Cyber, the School of Engineering and Information technology (SEIT), UNSW Canberra @ the Australian Defence Force Academy (ADFA). The datasets were gathered in a parallel processing to collect several normal and cyber-attack events from IoT networks.

#### 3.3.1 Feature Selection and Data Pre-processing

To ensure a direct comparison with my dataset, feature selection was performed similarly for both datasets. Having access to the pcapng files of the TON\_IoT dataset, the same features were selected. The data transformation and cleaning process was also identical for both datasets: numeric data were converted to float64, and filters were applied to eliminate lesser-used protocols, keeping only the most representative protocols.

#### 3.3.2 Number of selected Samples

The main difference between my dataset and the TON\_IoT dataset concerns the number of samples and in the absence of MQTT traffic (important for IoT world). The TON\_IoT



---

### 3. CREATION OF A REAL IOT ATTACK DATASET

---

dataset includes a large number of samples distributed among different classes of traffic, both benign and malicious, as shown in Table 3.5:

Class	# Samples	Percentage (%)
Benign	31,205,096	16.92
Backdoor	2,094,278	1.14
DDoS	48,803,148	26.47
DoS	27,258,952	14.78
Injection	3,534,193	1.92
MITM	10,324,322	5.60
Password	6,344,791	3.44
Ransomware	2,255,968	1.22
Scanning	30,109,397	16.33
XSS	22,472,425	12.19
<b>Total</b>	<b>184,402,570</b>	<b>100.00</b>

Table 3.5: Distribution of samples by class in the TON\_IoT dataset.

In the case of the TON\_IoT dataset, the minimum number of samples for a class is **2,094,278**. However, to ensure balance and parity of comparison with my dataset, I decided to select **34,601 samples** for each class, for a total of 4 classes as shown in Table 3.6:

Class	# Samples	Percentage (%)
Benign	34,601	25.00
DoS	34,601	25.00
Password	34,601	25.00
Scanning	34,601	25.00
<b>Total</b>	<b>138,404</b>	<b>100.00</b>

Table 3.6: Balanced TON\_IoT dataset.

In the balancing process, the *Password* class was renamed to `brute_force` to more accurately represent the type of attack, while the *Scanning* class was renamed to `recon` to highlight the nature of the recognition traffic.

#### 3.3.3 Summary of Datasets Differences

The Table 3.7 provides a clear and concise overview of the similarities and differences between my dataset and the TON\_IoT dataset, highlighting the number of classes, the classes themselves, and the total number of samples. Additionally, it notes that the selected features are identical for both datasets.

Dataset	# Classes	Classes	Total # Samples
My Dataset	4	Benign, DoS, Recon, Brute Force	138,404 (34,601 per class)
My Dataset	5	Benign, DoS, Recon, Brute Force, MQTT	173,005 (34,601 per class)
TON_IoT	4	Benign, DoS, Recon, Brute Force	138,404 (34,601 per class)
<b>Note:</b> The selected features are the same for both datasets.			

Table 3.7: Comparison of the used datasets.

## Chapter 4

# Construction of an Intrusion Detection System

In this chapter, the results of experiments conducted in order to compare the effectiveness of traditional Machine Learning (ML) models with Siamese network are detailed. The main objective is to determine which of the two approaches offers superior performance in the specific research context. To achieve this, several experiments were carried out using two separate datasets: one independently developed and the public TON\_IoT dataset. The experiments focused on several crucial aspects, including the performance evaluation of the models, the implementation of transfer learning and the generalization capability of the created dataset. In particular, transfer learning was applied using the TON\_IoT dataset, employing a pre-trained network on the internally developed dataset. This approach made it possible to exploit the knowledge gained from the model on the proprietary dataset, potentially improving performance on the public dataset and facilitating the adaptation of the model to new data.

Another fundamental aspect of the experiments concerns checking the adequacy of the dataset developed in-house to ensure that the results obtained were representative and could also be adopted by the scientific community. This step is essential to ensure the validity and replicability of the experiments, demonstrating that the dataset possesses a robust capacity for generalization and can be used as a reference in the existing literature. The chapter goes on to describe the results obtained from the different experiments are presented, accompanied by a comparative analysis between traditional ML models and Siamese networks. Such comparisons make it possible to clearly identify which approach proves to be more effective in terms of accuracy, efficiency and generalization capability, thus providing a solid basis for future conclusions and recommendations.

Through this chapter, I provide a comprehensive and rigorous overview of the methodologies adopted and the results achieved, contributing significantly to the validation of the research hypotheses and the progress of the study undertaken.

### 4.1 Data Pre-processing

The data preprocessing phase was meticulously executed to ensure the suitability and consistency of the datasets employed in both the traditional Machine Learning (ML) models and the Siamese network. This process was uniformly applied to the two distinct datasets utilized in the study: the proprietary dataset developed autonomously and the publicly available TON\_IoT dataset. Notably, for the ML models, a validation set was not employed, in contrast to the approach adopted for the Siamese network. The following section describes in detail the preprocessing operations divided into three subsections, explaining the use of the two main functions (`load_and_preprocess_data` and `preprocess_dataset`) and analysing the code blocks implemented.

#### 4.1.1 Data Cleaning

The first step in the preprocessing process involved cleaning the data in order to remove any anomalies and ensure the integrity of the dataset. This was achieved through the following steps.

##### Loading and Splitting the dataset

The function `load_and_preprocess_data` was used to load data from CSV files using the semicolon (;) as a delimiter.

```
df = pd.read_csv(csv_file, sep=";")
```

After loading, the dataset was subdivided into training, validation and testing subsets for the Siamese network, while for the Machine Learning (ML) models, a subdivision was made into training and testing only. In both subdivisions, a stratified split procedure based on the feature `type_attack` was adopted in order to ensure that the distribution of the different attack types remained consistent across the various subsets.

```
# Split Dataset into train/test for ML classifiers
train_df, test_df = train_test_split(df, test_size=test_size, stratify=df['type_attack'],
    ↪ shuffle=True)

# Split Dataset into train+val/test for Siamese net
train_val_df, test_df = train_test_split(df, test_size=test_size, stratify=df['type_attack'],
    ↪ shuffle=True)

train_df, val_df = train_test_split(train_val_df, test_size=val_size,
    ↪ stratify=train_val_df['type_attack'], shuffle=True)
```

In particular, for the ML models, 80% of the data was allocated to the training set and 20% to the testing set. In contrast, a more detailed subdivision was adopted for the Siamese network, with 64% of the data allocated to training, 16% to validation and 20% to

testing. This subdivision makes it possible to optimize the training of the Siamese network through the use of the validation set for monitoring and adjusting model parameters, further improving the generalization capability of the model to testing data.

### Removal of Not Validated Values

The `preprocess_dataset` function was invoked to perform the cleaning of protocol data. A default mapping (`protocol_map`) was defined to convert protocol names into numeric identifiers. Lines containing protocol types not present in this mapping were identified and removed, as shown in the following code block:

```
protocol_map = {"TCP": 1, "UDP": 2, "ICMP": 3, "ARP": 4, "MQTT": 5, "SNA": 6, "SSH": 7,
               "SSHv2": 8, "HPEXT": 9, "DNS": 10, "WiMax": 11, "NTP": 12, "0xc0a8": 13,
               "HTTP": 14, "TLSv1": 15, "WebSocket": 16, "TLSv1.2": 17, "RPCAP": 18,
               "HTTP/XML": 19, "SMTP": 20, "HTTP/JSON": 21, "SSDP": 22, "FTP": 23,
               "TLSv1.1": 24, "MDNS": 25, "SSLv3": 26, "IMAP": 27, "IGMPv2": 28, "POP": 29,
               "SMB": 30, "LLMNR": 31, "NBNS": 32, "RDP": 33, "IGMPv3": 34, "WHOIS": 35,
               "BROWSER": 36, "FTP-DATA": 37, "TLSv1.3": 38, "SMB2": 39, "DCERPC": 40,
               "EPM": 41, "OCSP": 42, "DHCPv6": 43, "BJNP": 44}

valid_protocols = protocol_map.keys()
valid_rows = df['Protocol_Type'].isin(valid_protocols)

num_removed = (~valid_rows).sum()
if num_removed > 0:
    print(f"Removed {num_removed} rows with not validated values in 'Protocol_Type'.")

df = df[valid_rows].reset_index(drop=True)
```

### Separation of Features and Target

After cleaning, the features ( $X$ ) and the target ( $y$ ) were separated. The `Protocol_Type` feature was transformed using the previously defined mapping:

```
X = df.drop(columns=['type_attack'])
y = df['type_attack']

X['Protocol_Type'] = X['Protocol_Type'].map(protocol_map)

if X['Protocol_Type'].isnull().any():
    raise ValueError("Some values of 'Protocol_Type' are not present in
    ↪ the map.")
```

This transformation ensures that all protocol types are encoded numerically, making them suitable for handling by ML models.

### 4.1.2 Data Transformation

Data transformation involved the encoding of target labels. This process was also handled by the `preprocess_dataset` function.

#### Target Label Encoding

To deal with the categorical nature of the target variable `type_attack`, label encoding was employed. During the training phase, a `LabelEncoder` was trained and subsequently used to transform the target label into a numeric format:

```
if train:
    label_encoder = LabelEncoder()
    y_encoded = label_encoder.fit_transform(y)
else:
    if label_encoder is None:
        raise ValueError("LabelEncoder non fornito per i dati di test.")
    y_encoded = label_encoder.transform(y)
```

For the testing phase, the same previously trained `LabelEncoder` was used to ensure consistency in label transformation, preventing discrepancies between training and testing labels.

### 4.1.3 Data Normalization

Normalization of the feature data was essential to ensure that all features contribute equally to the model learning process. This was performed using the function `MinMaxScaler` of *scikit-learn*. The steps performed are described below.

#### Application of the MinMaxScaler

In the training subset, the scaler was trained on the data and then used to transform the feature values, scaling them to a range between 0 and 1:

```
if train:
    scaler = MinMaxScaler()
    X_processed = scaler.fit_transform(X)
else:
    if scaler is None:
        raise ValueError("Preprocessor non fornito per i dati di test.")
    X_processed = scaler.transform(X)
```

This block of code checks whether the data is being trained. If so, an instance of `MinMaxScaler` is created, trained on the training data and used to transform the characteristics.

For the test data, the same scaler trained on the training data is used to ensure the consistency of the transformations.

### Data Type Conversion

All processed feature data were subsequently converted to the `float32` data type to facilitate efficient calculation during training and model evaluation:

```
X_processed = X_processed.astype("float32")
```

### Final Preprocessing Output

The final output of the preprocessing pipeline included the transformed feature matrices (`X_train` and `X_test`), the encoded target vectors (`y_train` and `y_test`), and the trained instances of the scaler and label encoder.

```
return X_train, y_train, X_test, y_test, scaler, label_encoder
```

## 4.2 Experimental Setup

This section will detail the methodologies and configurations adopted to conduct the experiments aimed at comparing traditional Machine Learning (ML) models and Siamese network. The main objective is to evaluate the effectiveness of each approach in the specific context of attack classification, using two distinct datasets: a proprietary one developed independently and the public TON\_IoT dataset.

### 4.2.1 Machine Learning Models

In this section, I will describe the traditional ML models used for classifying attacks in the context of the study and their configuration. Three classifiers were chosen: *Support Vector Machine* (SVM), *Random Forest* (RF) and *K-Nearest Neighbours* (KNN). The selection of these models is motivated by their proven effectiveness in classification tasks, the diversity of their architectures and the different learning strategies they employ, thus allowing for a robust and in-depth comparison of their performance in the specific context of the research.

The choice of SVM, RF and KNN is further motivated by their popularity and effectiveness in the construction of intrusion detection systems, as evidenced in several studies [46]. These models offer a diverse mix of approaches, ranging from maximum separation margins in SVM to tree-based ensembles in RF and similarity-based learning in KNN.

### 4.2.1.1 Support Vector Machine (SVM)

The SVM is a supervised classification model known for its ability to handle high-dimensionality problems and to find an optimal decision boundary between classes. Its effectiveness in handling non-linear data through the use of kernels makes it particularly suitable for complex classification tasks.

```
# Initialise SVM classifier
svm_classifier = SVC()

# Train SVM classifier
svm_classifier.fit(X_train, y_train)
```

In the context of this study, the SVM was implemented using the Radial Basis Function (RBF) kernel with default parameters, which allow the non-linearity of the data to be handled effectively. The main features of the SVM include finding the maximum margin between classes, using support vectors to define the optimal decision boundary.

### 4.2.1.2 Random Forest (RF)

The Random Forest is an ensemble of decision trees, which aggregates the results of many trees to improve accuracy and prevent overfitting. Its ability to handle large datasets with numerous features and to provide estimates of the importance of variables makes it a powerful tool for classification.

```
# Initialise RF classifier
rf_classifier = RandomForestClassifier(n_estimators=100)

# Train RF classifier
rf_classifier.fit(X_train, y_train)
```

In the present study, the Random Forest classifier was configured with `n_estimators=100`, indicating the number of trees in the forest. A larger number of trees tends to improve the accuracy of the model while reducing the variance. Each decision tree is trained on a random subset of the data and features, promoting diversity between trees and improving the robustness of the overall model.

### 4.2.1.3 K-Nearest Neighbors (KNN)

The KNN is a distance-based classification algorithm that assigns a class to a point based on the majority of classes of its nearest neighbors. Its simplicity make it ideal for datasets where the local structure of the data is significant.



```
# Initialise KNN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=5)

# Train KNN classifier
knn_classifier.fit(X_train, y_train)
```

In this study, the KNN classifier was configured with `n_neighbors=5`, which specifies the number of neighbors to be considered for classification. A value of 5 balances the sensitivity and stability of the model. KNN assigns the class based on the majority of the classes of the 5 nearest neighbors by calculating the distances between the points in a normalized feature space.

### 4.2.1.4 Common Procedures for Machine Learning Classifiers

All three classifiers used (SVM, RF and KNN) share common implementation components, such as data *preprocessing*, *model saving* and *result printing*. In fact, the `train` function will be presented below, highlighting the shared parts.

#### Preprocessing Data

Initially, the `load_and_preprocess_data` function (described in 4.1.1) is called, which takes care of loading the dataset from the CSV file, splitting it into training and testing sets, and applying the label cleaning, normalization and encoding operations:

```
def train(csv_file, model_filename, train):
    if train:
        # Load and preprocess the dataset
        X_train, y_train, X_test, y_test, scaler, label_encoder =
            ↪ load_and_preprocess_data(csv_file, test_size=0.2)

        ...
```

#### Save Trained Model

After training the classifier, model and preprocessing objects (`scaler` and `label_encoder`) are saved to disk using the *pickle module*. This allows them to be reloaded in the future without having to repeat the preprocessing or training operations:

```
# Save trained model, scaler and label encoder
with open(model_filename, 'wb') as file:
    pickle.dump((classifier, scaler, label_encoder), file)
print(f"Model and pre-processing saved as '{model_filename}'.")
```

### Evaluation on the Training Set

After saving the model and preprocessing objects, a prediction is made on the training set using the trained model. Subsequently, evaluation metrics (Accuracy, Precision, Recall, F1-Score) are calculated to measure the performance of the model on the training set. This process aims to identify possible overfitting problems.

```
# Evaluation on the Training Set
y_train_pred = classifier.predict(X_train)

# Calculate training evaluation metrics
train_accuracy = accuracy_score(y_train, y_train_pred)
train_precision = precision_score(y_train, y_train_pred, average='macro', zero_division=0)
train_recall = recall_score(y_train, y_train_pred, average='macro', zero_division=0)
train_f1 = f1_score(y_train, y_train_pred, average='macro', zero_division=0)

# Print results on training
print("\nResults on Training:")
print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Training Precision: {train_precision:.4f}")
print(f"Training Recall: {train_recall:.4f}")
print(f"Training F1 Score: {train_f1:.4f}")
```

### Evaluation on the Testing Set

If the train mode is set to False, the function loads the previously saved model and preprocessing objects. It then loads the test dataset from the CSV file and applies preprocessing using the same scaler and label encoder.

```
else:
    with open(model_filename, 'rb') as file:
        classifier, scaler, label_encoder = pickle.load(file)
    print(f"Modello caricato da '{model_filename}'.")

    df = pd.read_csv(csv_file, sep=";")

    # PRE-PROCESSING
    X_test, y_test, _, _ = preprocess_dataset(df, train=False,
        ↪ scaler=scaler, label_encoder=label_encoder)
```

Afterwards, a prediction is made on the test set using the trained model and the same evaluation metrics are calculated as for the training set.

```
# Evaluation on the Testing Set
y_pred = classifier.predict(X_test)

# Calculate testing evaluation metrics
test_accuracy = accuracy_score(y_test, y_pred)
test_precision = precision_score(y_test, y_pred, average='macro', zero_division=0)
test_recall = recall_score(y_test, y_pred, average='macro', zero_division=0)
```

```
test_f1 = f1_score(y_test, y_pred, average='macro', zero_division=0)

# Print results on testing
print("\nResults on Testing:")
print(f"Testing Accuracy: {test_accuracy:.4f}")
print(f"Testing Precision: {test_precision:.4f}")
print(f"Testing Recall: {test_recall:.4f}")
print(f"Testing F1 Score: {test_f1:.4f}")
```

Finally, the confusion matrix is calculated, which provides a visual representation of the correct predictions and errors made by the model. In addition, a classification report is generated that includes accuracy, recall and F1-score for each class.

```
# Calculate and print the confusion matrix on the test
cmatrix = confusion_matrix(y_test, y_pred)
print("\nTest Confusion Matrix:")
print(cmatrix)

# Print classification report
class_names = label_encoder.classes_
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=class_names, zero_division=0))
```

### 4.2.2 Siamese Neural Network

Siamese networks represent a particularly effective neural network architecture for comparison and verification tasks between input pairs. First introduced by Bromley et al. [47], these networks are designed to learn a similarity metric to determine how similar or dissimilar two instances are. The basic architecture of a Siamese network consists of two identical branches, each processing one of the two inputs. These branches share the same weights and parameters, ensuring that both instances are transformed into the same feature space.

#### 4.2.2.1 Generation of Pairs

A crucial element in the training of Siamese networks is the generation of data pairs. Pairs consist of two instances, which can be either similar (positive pairs) or dissimilar (negative pairs). Positive pairs are made up of data belonging to the same class, while negative pairs are made up of data belonging to different classes. This balancing of positive and negative pairs is essential to avoid bias in the model and to ensure that the network learns to effectively distinguish between similarity and dissimilarity [48].

#### Initialization and Data Preparation

In the present study, the function `generate_balanced_pairs` is designed to generate balanced pairs of data, respecting the balance between positive and negative pairs.

---

## 4. CONSTRUCTION OF AN INTRUSION DETECTION SYSTEM

---

At the beginning of the function, the `pairs` and `pair_labels` lists are initialized to store the data pairs and their similarity labels. A `generated_pairs` set is also created to keep track of the already generated pairs, avoiding duplications. In addition, a `class_indices` map is constructed that associates each class with its respective indices in the dataset, facilitating the sampling of pairs.

```
# Initialize pairs and labels
pairs = []
pair_labels = []

# Set to keep track of generated pairs
generated_pairs = set()

# Counters for duplicates
duplicate_attempts = 0

# Get unique classes from labels
unique_classes = np.unique(labels)
print(f"Classes: {len(unique_classes)}")

# Create a dictionary to hold indices for each class
class_indices = {label: np.where(labels == label)[0] for label in
    ↪ unique_classes}
```

### Calculation of the Number of Positive and Negative Pairs

The function also calculates the desired number of positive and negative pairs, ensuring that the total number of pairs is equal to `num_pairs`. This balancing is essential to maintain the balance between classes during training.

```
# Calculate number of positive and negative pairs
num_positive_pairs = num_pairs // 2
num_negative_pairs = num_pairs - num_positive_pairs
```

### Generation of Positive Couples

Positive pairs are generated by randomly selecting a class and choosing two instances belonging to that class. The function checks that the chosen class has at least two instances and that the generated pair has not already been created previously via the function `make_pair_key`. Each positive pair is labeled with a value of 1, indicating that the two instances belong to the same class. To avoid an infinite loop in the event that all required pairs cannot be generated, a maximum number of attempts is set (`max_attempts`). If

---

## 4. CONSTRUCTION OF AN INTRUSION DETECTION SYSTEM

---

the number of generated pairs does not reach the desired number, a warning message is issued.

```
def make_pair_key(idx1, idx2):
    return tuple(sorted((int(idx1), int(idx2))))

# Maximum number of failed attempts to prevent infinite loops
max_attempts = num_pairs * 1000

# Generate positive pairs
positive_pairs_generated = 0
attempts = 0 # Reset attempts for positive pairs
while positive_pairs_generated < num_positive_pairs and attempts <
↳ max_attempts:
    # Select a random class
    class_label = random.choice(unique_classes)
    # Check if there are at least 2 samples in this class
    if len(class_indices[class_label]) < 2:
        attempts += 1
        continue
    idx1, idx2 = np.random.choice(class_indices[class_label], size=2,
↳ replace=False)
    pair_key = make_pair_key(idx1, idx2)
    if pair_key in generated_pairs:
        attempts += 1
        duplicate_attempts += 1
        continue
    # Add the pair and label
    pairs.append([data[idx1], data[idx2]])
    pair_labels.append(1) # Same class
    generated_pairs.add(pair_key)
    positive_pairs_generated += 1
    attempts = 0 # Reset attempts since we successfully added a pair

if attempts >= max_attempts:
    print("Reached maximum attempts while generating positive pairs.")
```

### Negative Pairs Generation

Similar to positive pairs, negative pairs are generated by randomly selecting two different classes and choosing one instance from each class. It is checked that the pairs are not duplicated and that the selected classes have at least one instance. Negative pairs are labeled with a value of 0, indicating that the two instances belong to different classes. Again, a maximum number of attempts is set (`max_attempts`) and if the number of generated pairs does not reach the desired number, a warning message is issued.

```
# Generate negative pairs
negative_pairs_generated = 0
attempts = 0 # Reset attempts for negative pairs
while negative_pairs_generated < num_negative_pairs and attempts <
↳ max_attempts:
    # Select two different classes
    if len(unique_classes) < 2:
        print("Not enough classes to generate negative pairs.")
        break
    class_label1, class_label2 = random.sample(list(unique_classes), 2)
    if len(class_indices[class_label1]) == 0 or
↳ len(class_indices[class_label2]) == 0:
        attempts += 1
        continue
    idx1 = np.random.choice(class_indices[class_label1])
    idx2 = np.random.choice(class_indices[class_label2])
    pair_key = make_pair_key(idx1, idx2)
    if pair_key in generated_pairs:
        attempts += 1
        duplicate_attempts += 1
        continue
    # Add the pair and label
    pairs.append([data[idx1], data[idx2]])
    pair_labels.append(0) # Different classes
    generated_pairs.add(pair_key)
    negative_pairs_generated += 1
    attempts = 0 # Reset attempts since we successfully added a pair

if attempts >= max_attempts:
    print("Reached maximum attempts while generating negative pairs.")
```

### Shuffle and Conversion to NumPy Arrays

Once all desired pairs have been generated, they are converted into *NumPy* arrays for more efficient handling during training. Next, the pairs are shuffled using the function `shuffle` to ensure a random distribution of pairs in the final dataset. The function returns the array of created pairs and the array with their respective labels.

```
pairs_array = np.array(pairs)
pair_labels = np.array(pair_labels)

pairs_array, pair_labels = shuffle(pairs_array, pair_labels,
↳ random_state=None)

return pairs_array, pair_labels
```

### 4.2.2.2 Neural Network Architecture

The architecture of the Siamese networks can be composed of two parallel branches, each of which includes a set of convolutional and fully connected layers. The convolutional layers are responsible for extracting features from the input data, while the fully connected layers aggregate these features to produce a compact and discriminative representation of each instance. At the end of the two branches, the pairwise representations are compared using a similarity function, typically based on Euclidean or Manhattan distance, to measure the similarity between the two instances [49].

In my study, the class `SiameseNet` defines the architecture of the Siamese network used. It accepts two shape-identical inputs (`input_shape`), represented by `left_input` and `right_input`, respectively. Both inputs are processed through a sequence of convolutional layers defined within the `Sequential` model called `convnet`.

```
self.left_input = Input(input_shape)
self.right_input = Input(input_shape)
self.convnet = Sequential()

self.convnet.add(Conv2D(filters=256, kernel_size=(50, 1), strides=(1, 1),
    ↪ activation='relu', padding='same', input_shape=input_shape))
self.convnet.add(Conv2D(filters=128, kernel_size=(10, 1), strides=(1, 1),
    ↪ activation='relu', padding='same'))
self.convnet.add(MaxPooling2D(pool_size=(2, 1)))

self.convnet.add(Conv2D(filters=128, kernel_size=(5, 1), strides=(1, 1),
    ↪ activation='sigmoid', padding='same'))
self.convnet.add(MaxPooling2D(pool_size=(2, 1)))

self.convnet.add(Conv2D(filters=64, kernel_size=(3, 1), strides=(1, 1),
    ↪ activation='sigmoid', padding='same'))
self.convnet.add(MaxPooling2D(pool_size=(2, 1)))

self.convnet.add(Conv2D(filters=32, kernel_size=(3, 1), strides=(1, 1),
    ↪ activation='sigmoid', padding='same'))
self.convnet.add(MaxPooling2D(pool_size=(2, 1)))

self.convnet.add(Flatten())
```

The structure of the `convnet` begins with a first `Conv2D` layer using 256 filters of size (50.1), followed by a second `Conv2D` layer with 128 filters of size (10.1). These layers are accompanied by activation functions `relu`, which introduce non-linearity and facilitate the learning of complex representations. Subsequently, a maximum pooling layer (`MaxPooling2D`) with a pooling window of (2,1) reduces the spatial dimensionality, helping to extract the most relevant features and decrease the risk of overfitting.

---

## 4. CONSTRUCTION OF AN INTRUSION DETECTION SYSTEM

---

Two further convolutional layers with 128 and 64 filters respectively, of size (5,1) and (3,1), are followed by further maximum pooling layers. These layers utilize the activation function `sigmoid`, which normalizes the output between 0 and 1, facilitating better information handling in subsequent model steps. The sequence of convolution and pooling continues with a further Conv2D layer of 32 filters and a further pooling layer, thus completing the convolutional part of the model. After convolutional operations, the data are flattened (`Flatten`), transforming activation maps into feature vectors that can be compared directly.

The convolutional network is applied to each of the two inputs (`left_input` and `right_input`), producing `encoded_l` and `encoded_r`, respectively. This process of sharing the weights between the two branches ensures that both paths of the network learn similar representations, which is crucial for the evaluation of similarity between the inputs.

```
self.encoded_l = self.convnet(self.left_input)
self.encoded_r = self.convnet(self.right_input)
```

The final part of the architecture involves the introduction of a `Lambda` layer that calculates the **Euclidean distance** between the representations learned from the two inputs (`encoded_l` and `encoded_r`). This distance is then used as the output of the Siamese network, which is defined using the `Model` of *Keras*, specifying the input and output.

```
self.L1_layer = Lambda(self.euclidean_distance,
    ↪ output_shape=self.eucl_dist_output_shape)

self.L1_distance = self.L1_layer([self.encoded_l, self.encoded_r])
self.siamese_net = Model(inputs=[self.left_input, self.right_input],
    ↪ outputs=self.L1_distance)
```

For training, the network uses the optimizer *Adam* with an initial learning rate of 0.0001, chosen for its ability to dynamically adapt the learning rate during optimization. The loss function adopted is **contrastive loss**, which is particularly suitable for Siamese networks because it encourages the model to reduce the distance between similar input pairs and increase it between dissimilar pairs. In addition, a metric of **accuracy** is used to monitor the performance of the model during training.

```
lr = 0.0001

self.optimizer = Adam(lr)
self.siamese_net.compile(loss=self.contrastive_loss, optimizer=self.optimizer,
    ↪ metrics=[self.accuracy])
```



```
def contrastive_loss(self, y_true, y_pred):  
    '''Contrastive loss from Hadsell-et-al.'06  
    http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf  
    '''  
    margin = 1  
    y_true = K.cast(y_true, 'float32')  
    sqaure_pred = K.square(y_pred)  
    margin_square = K.square(K.maximum(margin - y_pred, 0))  
    return K.mean(y_true * sqaure_pred + (1 - y_true) * margin_square)
```

The function accuracy transforms the calculated continuous distances between input pairs into binary predictions by applying a fixed threshold (in this case, 0.5). If the predicted distance is less than this threshold, the pair is classified as similar (1), otherwise as dissimilar (0). Next, these binary predictions are compared with the actual labels (`y_true`) to determine the proportion of correct classifications.

```
def accuracy(self, y_true, y_pred):  
    '''Compute classification accuracy with a fixed threshold on  
    distances.  
    '''  
    return K.mean(K.equal(y_true, K.cast(y_pred < 0.5, y_true.dtype)))
```

### 4.2.2.3 Concept of Similarity and Distance

The core of Siamese networks lies in the ability to measure the similarity between two instances through a distance function. This function calculates how close or far apart two feature representations are in feature space, thus determining the similarity between the two instances. The goal of training is to minimize the distance between positive pairs and maximize the distance between negative pairs, allowing the network to learn a discriminative metric [47]. This similarity concept is crucial for applications such as facial recognition, signature verification and, in the context of this project, attack classification. In fact, the function `euclidean_distance` precisely implements the concept of measuring the distance between two feature representations. The function takes two vectors (`x` and `y`), calculates the sum of the squares of the differences between their corresponding components and returns the square root, ensuring that the result is not less than a small constant (`K.epsilon()`) to avoid numerical problems.

```
def euclidean_distance(self, vects):  
    x, y = vects  
    sum_square = K.sum(K.square(x - y), axis=1, keepdims=True)  
    return K.sqrt(K.maximum(sum_square, K.epsilon()))
```

Whereas, the function `eucl_dist_output_shape` defines the shape of the output of the Euclidean distance function, ensuring that the result is compatible with the model's expectations. This is important when using a Lambda layer in Keras, since Keras needs to know the shape of the output to correctly construct the computational graph.

```
def eucl_dist_output_shape(self, shapes):  
    shape1, shape2 = shapes  
    return (shape1[0], 1)
```

### 4.2.2.4 Train of Siamese Network

In the study undertaken, the training phase of the Siamese network is performed through a series of well-structured steps that optimize the performance of the model. Initially, the Siamese model is created by instantiating the `SiameseNet` class with an appropriate input shape derived from the training data dimensions (`X_train.shape[1]`). This step configures the network architecture with the previously defined convolutional layers ready to process the input pairs.

```
siamese_model = (SiameseNet(input_shape=(X_train.shape[1], 1, 1))).get()
```

To improve model generalization and prevent overfitting, *EarlyStopping callback* is used. This mechanism monitors the loss on the validation set (`val_loss`) and stops training if no improvement is observed for 10 consecutive epochs (`patience=10`). In addition, the `restore_best_weights=True` option ensures that the model weights are restored to the optimal values achieved during training, thus ensuring the best performance.

```
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
```

The actual training takes place via the `fit` method, which receives the training input pairs (`train_a` and `train_b`) together with their respective labels (`train_labels`). A batch size of 256 is specified, representing the number of samples processed before updating the model weights. This value balances computational efficiency and memory usage, allowing the model to learn stably without consuming excessive resources. The maximum number of epochs is set to 100, giving the model ample opportunity to learn. However, thanks to early stopping, training can stop earlier if no further improvement in validation loss is detected.

```
siamese_model.fit([train_a, train_b], train_labels,  
                  validation_data=([val_a, val_b], val_labels),  
                  batch_size=256,  
                  epochs=100,  
                  callbacks=[early_stopping])
```

During training, validation data (`val_a` and `val_b` with their respective `val_labels`) are also used to monitor the model's performance on unseen data, allowing continuous evaluation of its ability to generalize. The use of this data helps to quickly identify if the model is starting to overfit the training data, allowing for early intervention through early stopping.

### 4.2.2.4 Motivation of the Siamese Neural Network in the Thesis

The adoption of Siamese networks is motivated by the need to demonstrate a better ability to generalize the dataset for the classification of attacks. Siamese networks are particularly suitable for scenarios in which a limited number of examples are available for certain classes, as they can learn similarity relationships even with little data [49]. Furthermore, their ability to compare pairs of instances makes it possible to identify attacks of new types, increasing the flexibility and effectiveness of the detection system. By integrating the Siamese networks with the created dataset and `TON_IoT`, a robust and discriminative feature representation can be exploited, significantly improving the overall performance of the model in recognizing and classifying various types of attacks.

### 4.2.3 Transfer Learning Approach

Transfer Learning is a machine learning technique that exploits the knowledge gained from a pre-trained model on one dataset to improve performance on another related dataset. This approach is particularly advantageous when limited amounts of data are available for the new task, as it reduces training time and improves the generalization of the model. Instead of starting training from scratch, the pre-trained model provides a solid base of feature representations that can be further refined for the new specific domain.

In the context of this project, I applied Transfer Learning to the Siamese network initially trained on my primary dataset and then adapted it to a second dataset, `TON_IoT`, with the aim of assessing how generic my original dataset could be and how capable it was of generalizing to different data. After completing the initial training phase on the primary dataset, the pre-trained model was saved and then reloaded to be reused on the new dataset. This process was performed through the following code:

```
siamese_model = (SiameseNet(input_shape=(X_train.shape[1], 1,
↪ 1))) .load_saved_model(model_pretrained + "siamese_model.h5")

early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

history = siamese_model.fit([train_a, train_b], train_labels,
                             validation_data=([val_a, val_b], val_labels),
                             batch_size=256,
                             epochs=40,
                             callbacks=[early_stopping])
```

At this stage, the pre-trained Siamese network is loaded using the `load_saved_model` function, which restores the optimal weights obtained during the initial training on the primary dataset. Subsequently, the model is further trained on the TON\_IoT dataset with a reduced number of epochs (40 compared to the previous 100) and a lower patience for early stopping (5 epochs instead of 10). These changes in the hyperparameters reflect the assumption that the model already starts from a robust base of feature representations, thus needing fewer updates to adapt to the new domain.

The main objective of this approach is to test the ability of my original dataset to generalize to another dataset of a similar but distinct nature. If the pre-trained model manages to maintain good performance even on the new dataset, this indicates that the learned feature representations are sufficiently generic and transferable. Conversely, if performance decreases significantly, it might suggest that my primary dataset contains too specific features that do not apply well to the new context.

The fundamental difference between the initial training phase and the Transfer Learning phase lies in the initialization of the weights and the amount of training required. During the first phase, the model is trained from scratch on the primary dataset, allowing the network to learn domain-specific feature representations. In the case of Transfer Learning, the model uses the pre-trained weights as a starting point, speeding up the adaptation process to the new dataset and potentially improving performance due to the already optimized representations. In addition, the use of fewer epochs and shorter patience in early stopping helps to maintain training efficiency while avoiding overfitting to the new dataset.

# Chapter 5

## Evaluation of Results

This chapter presents an in-depth analysis of the experimental results obtained from the various classification methodologies applied throughout the project. The chapter aims to evaluate the performance of different approaches, including traditional machine learning classifiers, a Siamese network architecture, and the application of Transfer Learning. By providing a detailed comparison between these techniques, I seek to determine which methodologies are most effective for the problem at hand and under which conditions they perform optimally.

In this I show the experiments conducted on two distinct datasets: the original dataset, meticulously curated for this project, and the widely used TON\_IoT dataset. The choice of these datasets allows for a comprehensive evaluation of the models in terms of their ability to generalize across different data distributions. Importantly, all these experiments were conducted with the primary objective of evaluating whether the original dataset created for this project could outperform the TON\_IoT dataset, with the hope that it could be used as a benchmark dataset in the literature.

The results presented in this chapter are divided into several sections: comparisons are drawn between models trained solely on the original dataset and those trained and tested on the TON\_IoT dataset, providing insights into the models' capacity for generalization and their robustness against varying data distributions. The chapter also highlights the benefits and limitations of each approach, discussing practical trade-offs that are critical for real-world deployments.

### 5.1 Performance of Machine Learning Models

In this section I will present and analyse the results obtained from the traditional classifiers used in the project, namely Support Vector Machines (SVM), Random Forest (RF) and K-Nearest Neighbors (KNN). The results will be divided into two separate parts to provide a comprehensive overview of the performance of the models in different training and testing contexts.

### 5.1.1 Train and Test Classifiers

The first part focuses on the classifiers trained and evaluated using my original dataset, examined in two different classification configurations: one with 4 classes (excluding the MQTT attack class) and another with 5 classes (including the MQTT attack class).

In following tables I show the classification reports and testing results by Dataset.

- My Dataset with 4 classes (benign, brute\_force, dos, recon):

#### KNN

Class	Precision	Recall	F1-Score	Support
benign	0.99	0.99	0.99	6920
brute_force	1.00	0.99	1.00	6920
dos	1.00	1.00	1.00	6920
recon	0.99	0.99	0.99	6921
<b>Accuracy</b>			0.99	27681
<b>Macro Avg</b>	0.99	0.99	0.99	27681
<b>Weighted Avg</b>	0.99	0.99	0.99	27681

#### Random Forest

Class	Precision	Recall	F1-Score	Support
benign	0.99	0.98	0.99	6920
brute_force	1.00	1.00	1.00	6920
dos	1.00	1.00	1.00	6920
recon	0.99	0.99	0.99	6921
<b>Accuracy</b>			0.99	27681
<b>Macro Avg</b>	0.99	0.99	0.99	27681
<b>Weighted Avg</b>	0.99	0.99	0.99	27681

#### SVM

Class	Precision	Recall	F1-Score	Support
benign	0.96	0.93	0.95	6920
brute_force	0.97	0.97	0.97	6920
dos	0.98	0.96	0.97	6920
recon	0.95	0.99	0.97	6921
<b>Accuracy</b>			0.97	27681
<b>Macro Avg</b>	0.97	0.97	0.97	27681
<b>Weighted Avg</b>	0.97	0.97	0.97	27681

Classifier	Accuracy	Precision	Recall	F1 Score
KNN	99.33%	99.33%	99.33%	99.33%
Random Forest	99.21%	99.21%	99.21%	99.21%
SVM	96.59%	96.61%	96.59%	96.58%

---

## 5. EVALUATION OF RESULTS

---

- My Dataset with 5 classes (benign, brute\_force, dos, mqtt\_attack, recon):

### KNN

Class	Precision	Recall	F1-Score	Support
benign	0.99	0.99	0.99	6920
brute_force	0.84	0.81	0.83	6920
dos	0.99	0.99	0.99	6920
mqtt_attack	0.82	0.84	0.83	6920
recon	0.99	0.99	0.99	6921
<b>Accuracy</b>			0.92	34601
<b>Macro Avg</b>			0.92	34601
<b>Weighted Avg</b>			0.92	34601

### Random Forest

Class	Precision	Recall	F1-Score	Support
benign	0.98	0.98	0.98	6920
brute_force	0.84	0.81	0.82	6920
dos	0.99	0.99	0.99	6920
mqtt_attack	0.81	0.84	0.82	6920
recon	0.99	0.99	0.99	6921
<b>Accuracy</b>			0.92	34601
<b>Macro Avg</b>			0.92	34601
<b>Weighted Avg</b>			0.92	34601

### SVM

Class	Precision	Recall	F1-Score	Support
benign	0.91	0.93	0.92	6920
brute_force	0.76	0.87	0.81	6920
dos	0.97	0.97	0.97	6920
mqtt_attack	0.87	0.73	0.80	6920
recon	0.95	0.94	0.95	6921
<b>Accuracy</b>			0.89	34601
<b>Macro Avg</b>			0.89	34601
<b>Weighted Avg</b>			0.89	34601

Classifier	Accuracy	Precision	Recall	F1 Score
KNN	92.49%	92.49%	92.49%	92.48%
Random Forest	92.29%	92.31%	92.29%	92.29%
SVM	88.85%	89.16%	88.85%	88.80%

---

## 5. EVALUATION OF RESULTS

---

Subsequently, the same classifiers are also trained and evaluated on the TON\_IoT dataset. This comparison aims to check the quality and generalizability of my dataset against an external dataset, using the same configuration of classes and same hyperparameters.

- Dataset TON\_IoT with 4 classes (benign, brute\_force, dos, recon):

### KNN

Class	Precision	Recall	F1-Score	Support
benign	0.97	0.96	0.96	6920
brute_force	0.88	0.91	0.89	6920
dos	0.96	0.96	0.96	6920
recon	0.93	0.90	0.92	6921
<b>Accuracy</b>			0.93	27681
<b>Macro Avg</b>	0.93	0.93	0.93	27681
<b>Weighted Avg</b>	0.93	0.93	0.93	27681

### Random Forest

Class	Precision	Recall	F1-Score	Support
benign	0.98	0.95	0.96	6920
brute_force	0.88	0.93	0.90	6920
dos	0.95	0.96	0.96	6920
recon	0.94	0.90	0.92	6921
<b>Accuracy</b>			0.94	27681
<b>Macro Avg</b>	0.94	0.94	0.94	27681
<b>Weighted Avg</b>	0.94	0.94	0.94	27681

### SVM

Class	Precision	Recall	F1-Score	Support
benign	1.00	0.89	0.94	6920
brute_force	0.69	0.85	0.76	6920
dos	0.88	0.97	0.92	6920
recon	0.89	0.69	0.78	6921
<b>Accuracy</b>			0.85	27681
<b>Macro Avg</b>	0.86	0.85	0.85	27681
<b>Weighted Avg</b>	0.86	0.85	0.85	27681

Classifier	Accuracy	Precision	Recall	F1 Score
KNN	93.25%	93.30%	93.25%	93.26%
Random Forest	93.56%	93.66%	93.53%	93.58%
SVM	85.04%	86.46%	85.04%	85.13%



### 5.1.2 Evaluating pre-trained Classifiers

The second part focuses on evaluating the generalization capability of the classifiers across different datasets. Specifically, the classifiers were trained on my dataset in the two configurations (4 and 5 classes) and tested on the TON\_IoT dataset. Conversely, the classifiers pre-trained on the TON\_IoT dataset are tested on my dataset with 4 classes.

- Classifiers trained on my dataset (4 classes) and tested on TON\_IoT

#### KNN

Class	Precision	Recall	F1-Score	Support
benign	0.07	0.09	0.08	34601
brute_force	0.26	0.39	0.31	34601
dos	0.02	0.00	0.00	34601
recon	0.48	0.60	0.53	34601
<b>Accuracy</b>			0.27	138404
<b>Macro Avg</b>	0.21	0.27	0.23	138404
<b>Weighted Avg</b>	0.21	0.27	0.23	138404

#### Random Forest

Class	Precision	Recall	F1-Score	Support
benign	0.08	0.15	0.10	34601
brute_force	0.19	0.22	0.20	34601
dos	0.01	0.00	0.00	34601
recon	0.69	0.58	0.63	34601
<b>Accuracy</b>			0.24	138404
<b>Macro Avg</b>	0.24	0.24	0.23	138404
<b>Weighted Avg</b>	0.24	0.24	0.23	138404

#### SVM

Class	Precision	Recall	F1-Score	Support
benign	0.05	0.10	0.06	34601
brute_force	0.21	0.27	0.24	34601
dos	0.01	0.00	0.00	34601
recon	0.48	0.17	0.25	34601
<b>Accuracy</b>			0.14	138404
<b>Macro Avg</b>	0.18	0.14	0.14	138404
<b>Weighted Avg</b>	0.18	0.14	0.14	138404

Classifier	Accuracy	Precision	Recall	F1 Score
KNN	26.96%	20.86%	26.96%	23.13%
Random Forest	23.97%	24.17%	23.97%	23.48%
SVM	13.51%	18.46%	13.51%	13.63%

---

## 5. EVALUATION OF RESULTS

---

- Classifiers trained on my dataset (5 classes) and tested on TON\_IoT

### KNN

Class	Precision	Recall	F1-Score	Support
benign	0.06	0.08	0.07	34601
brute_force	0.21	0.29	0.24	34601
dos	0.02	0.00	0.00	34601
mqtt_attack	0.00	0.00	0.00	0
recon	0.54	0.58	0.56	34601
<b>Accuracy</b>			0.24	138404
<b>Macro Avg</b>	0.17	0.19	0.17	138404
<b>Weighted Avg</b>	0.21	0.24	0.22	138404

### Random Forest

Class	Precision	Recall	F1-Score	Support
benign	0.07	0.14	0.09	34601
brute_force	0.17	0.19	0.18	34601
dos	0.01	0.00	0.00	34601
mqtt_attack	0.00	0.00	0.00	0
recon	0.76	0.58	0.66	34601
<b>Accuracy</b>			0.23	138404
<b>Macro Avg</b>	0.20	0.18	0.19	138404
<b>Weighted Avg</b>	0.25	0.23	0.23	138404

### SVM

Class	Precision	Recall	F1-Score	Support
benign	0.07	0.08	0.07	34601
brute_force	0.21	0.26	0.23	34601
dos	0.01	0.00	0.00	34601
mqtt_attack	0.00	0.00	0.00	0
recon	0.42	0.58	0.49	34601
<b>Accuracy</b>			0.23	138404
<b>Macro Avg</b>	0.14	0.18	0.16	138404
<b>Weighted Avg</b>	0.18	0.23	0.20	138404

Classifier	Accuracy	Precision	Recall	F1 Score
KNN	23.69%	16.57%	18.95%	17.44%
Random Forest	22.81%	20.25%	18.25%	18.62%
SVM	22.82%	14.09%	18.26%	15.78%

---

## 5. EVALUATION OF RESULTS

---

- Classifiers trained on TON\_IoT and tested on my dataset (4 classes)

### KNN

Class	Precision	Recall	F1-Score	Support
benign	0.89	0.08	0.07	34601
brute_force	0.38	0.29	0.24	34601
dos	0.00	0.00	0.00	34601
recon	0.37	0.61	0.46	34601
<b>Accuracy</b>			0.42	138404
<b>Macro Avg</b>			0.39	138404
<b>Weighted Avg</b>			0.39	138404

### Random Forest

Class	Precision	Recall	F1-Score	Support
benign	0.83	0.56	0.67	34601
brute_force	0.44	0.56	0.49	34601
dos	0.00	0.00	0.00	34601
recon	0.18	0.35	0.23	34601
<b>Accuracy</b>			0.37	138404
<b>Macro Avg</b>			0.35	138404
<b>Weighted Avg</b>			0.35	138404

### SVM

Class	Precision	Recall	F1-Score	Support
benign	0.95	0.02	0.05	34601
brute_force	0.50	0.95	0.66	34601
dos	0.00	0.00	0.00	34601
recon	0.41	0.74	0.53	34601
<b>Accuracy</b>			0.43	138404
<b>Macro Avg</b>			0.31	138404
<b>Weighted Avg</b>			0.31	138404

Classifier	Accuracy	Precision	Recall	F1 Score
KNN	42.32%	41.00%	42.32%	38.84%
Random Forest	36.83%	36.02%	36.83%	34.83%
SVM	43.00%	46.76%	43.00%	30.91%

### 5.1.3 Report of results obtained

In order to evaluate the performance of the three classifiers (KNN, Random Forest, SVM) trained on the mentioned datasets and to understand from an initial analysis which of the two datasets (the one created or TON\_IoT) presents better aspects, a complete analysis of the accuracy, precision, recall and F1 score metrics was conducted. From the results presented, I can draw the following conclusions:

**1. Performance on the Dataset created with 4 classes:**

- **KNN** performed excellently with an accuracy of 99.33%, precision, recall and F1 scores equivalent (all 99.33%). Analysis of the confusion matrix also shows few errors evenly distributed among the classes, indicating a good ability to distinguish the different types of attack.
- **Random Forest** performed very well, with slightly lower accuracy than KNN, at 99.21%. The confusion matrix indicates that errors are more concentrated in the benign and recon classes, but with very small values, still denoting a high classification capacity.
- **SVM** showed a lower accuracy than the other two classifiers, standing at 96.59%. SVM had more difficulty distinguishing between benign and dos classes than the other two classifiers. However, the F1 score of 96.58% still shows good generalisation capabilities.

**2. Performance on Dataset created with 5 classes:** with the increase in the number of classes to 5 (addition of the *mqtt\_attack* class), all classifiers showed a decrease in performance.

- **KNN** achieved an accuracy of 92.49%, with similar accuracy, recall and F1 scores. The confusion matrix shows that the *mqtt\_attack* and *brute\_force* class were more problematic to classify.
- **Random Forest** achieved an accuracy of 92.29%, similar to KNN, but slightly lower in precision and recall, especially for *brute\_force* and *mqtt\_attack*.
- **SVM** showed an accuracy of 88.85%, showing greater difficulty in distinguishing *mqtt\_attack* and *brute\_force*. This is evident in the confusion matrix, where these classes have a significant number of misclassifications.

**3. Performance on the TON\_IoT Dataset with 4 classes:** the results obtained on the TON\_IoT dataset are inferior to the dataset created, but still demonstrate a good ability to generalize the models.

- **KNN** achieved an accuracy of 93.25%, with similar precision and recall. The confusion matrix shows a good ability to classify the benign, *brute\_force*, dos, and recon classes, with most errors between *brute\_force* and recon.

- **Random Forest** outperformed KNN with an accuracy of 93.56%, showing greater stability in accuracy, recall and F1 scores.
  - **SVM** had a significant drop in performance with an accuracy of 85.04%. The confusion matrix shows particular difficulties in distinguishing *brute\_force* and *recon*, with many false positives and false negatives.
4. **Performance of models trained on the created Dataset (4 Classes) and tested on TON\_IoT:** all classifiers showed a strong reduction in performance when tested on the TON\_IoT dataset after being trained on the created dataset. This indicates poor generalization between the two datasets. The best classifier in this case was the KNN with an accuracy of 26.96%, followed by the Random Forest (23.97%) and finally the SVM (13.51%).
  5. **Performance of the trained models on the Dataset created (5 Classes) and tested on TON\_IoT:** the results for the 5 class version of the dataset created trained and then tested on TON\_IoT indicate very weak performance for all classifiers, with accuracies varying between 22.69% and 23.82%. This highlights the difficulty of predicting the new *mqtt\_attack* class in the context of the TON\_IoT dataset, which is absent, leading to very low performance.
  6. **Performance of the trained models on TON\_IoT dataset and tested on Dataset created (4 classes):** the pre-trained classifiers on TON\_IoT showed limited generalization ability on my 4-class dataset, with maximum accuracy of 43% (SVM) and difficulty in recognizing some classes, such as *dos*. The *benign* and *brute\_force* classes achieved better metrics, but the low f1-score macro values highlight the need for further optimization to improve fit between different datasets.

Considering all datasets and evaluation metrics, Random Forest is the classifier with the best overall performance. It demonstrated good accuracy on both the trained datasets and in tests, with greater stability than the other classifiers. It also had a high F1 score on all classes, showing a good ability to balance accuracy and recall, especially in situations where the other classes were poorly represented.

On the other hand, it can be seen that the dataset created is well balanced between the classes, whereas, the TON\_IoT dataset has disparities in the distribution of the classes and contains some ‘difficult’ examples (i.e. examples that are less distinctive or noisier), causing a deterioration in performance. Therefore, the TON\_IoT dataset proved to be more robust and challenging for the models, being superior in terms of complexity and generalization capability. The dataset created, on the other hand, performed very well in its own domain, but showed significant limitations when tested on a different context, suggesting that it is less complex or less representative than TON\_IoT.

For these reasons, further tests were conducted exploiting neural networks, in particular, a Siamese network was used.

## 5.2 Performance of Siamese Networks

The following section presents and analyzes the performance results of the Siamese network. The primary objective of these experiments was to determine whether the dataset created for this project could be considered superior or comparable to the TON\_IoT dataset, with the hope that it might be used as a benchmark by the broader research community. In particular, I will show and discuss the outcomes obtained from using the Siamese network on the dataset created for this project with both 4 and 5 classes, as well as on the TON\_IoT dataset. For each dataset, the Siamese network was trained on 1,000,000 pairs of data for training and validation, and tested on 500,000 pairs.

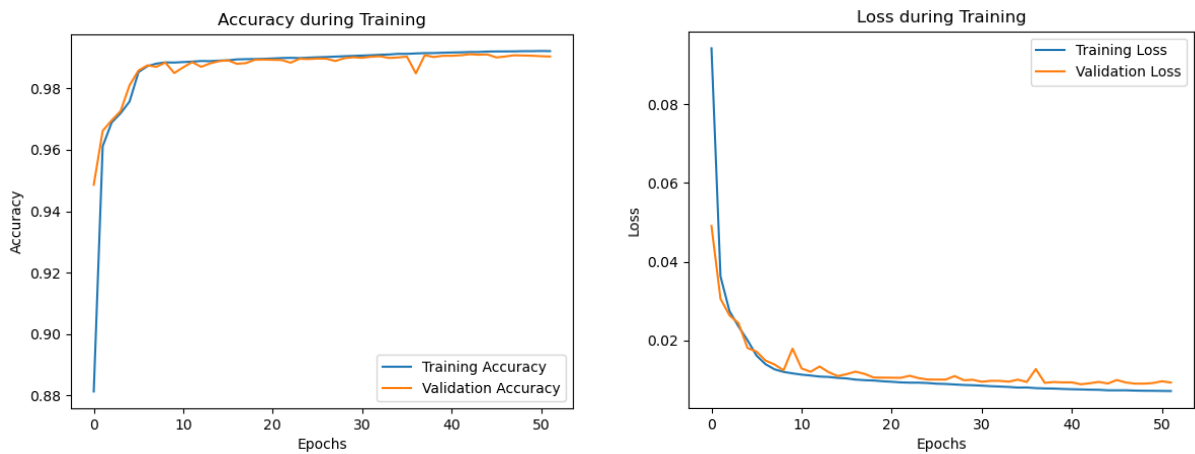
### 5.2.1 Results of my Dataset with 4 Classes

The results obtained in Table 5.1 with the Dataset created with 4 classes show an **accuracy** on the test set of **99.15%** and a **loss** of **0.0079**. The other classification metrics (Accuracy, Recall and F1-score), for both classes, are close to 99%.

Class	Precision	Recall	F1-Score	Support
0	0.99	1.00	0.99	150000
1	1.00	0.99	0.99	150000
<b>Accuracy</b>			0.99	300000
<b>Macro Avg</b>	0.99	0.99	0.99	300000
<b>Weighted Avg</b>	0.99	0.99	0.99	300000

Table 5.1: Classification Report Dataset with 4 Classes.

Below in Figure 5.1 I will show the plot of accuracy and loss in the training phase.



(a) Plot Training Accuracy.

(b) Plot Training Loss.

Figure 5.1: Plot of my Dataset with 4 Classes.

---

## 5. EVALUATION OF RESULTS

---

In this dataset with 4 classes, the curves show a very rapid increase in accuracy (Figure 5.1 (a)), reaching 99% in the first epochs and remaining stable throughout the training. The final loss (Figure 5.1 (b)) is significantly lower (0.0079), with rapid stabilization and no fluctuations, indicating that the model succeeds in classifying the classes efficiently and without signs of overfitting. This suggests that the dataset is well balanced and highly representative, allowing the model to clearly distinguish between classes.

### 5.2.2 Results of my Dataset with 5 Classes

The results obtained in Table 5.2 with the Dataset created with 5 classes show an **accuracy** on the test set of **95.02%** and a **loss** of **0.0393**. The other classification metrics (Accuracy, Recall and F1-score), for both classes, are close to 95%.

Class	Precision	Recall	F1-Score	Support
0	0.97	0.93	0.95	150000
1	0.93	0.97	0.95	150000
<b>Accuracy</b>			0.95	300000
<b>Macro Avg</b>			0.95	300000
<b>Weighted Avg</b>			0.95	300000

Table 5.2: Classification Report Dataset with 5 Classes.

Below in Figure 5.2 I will show the plot of accuracy and loss in the training phase.

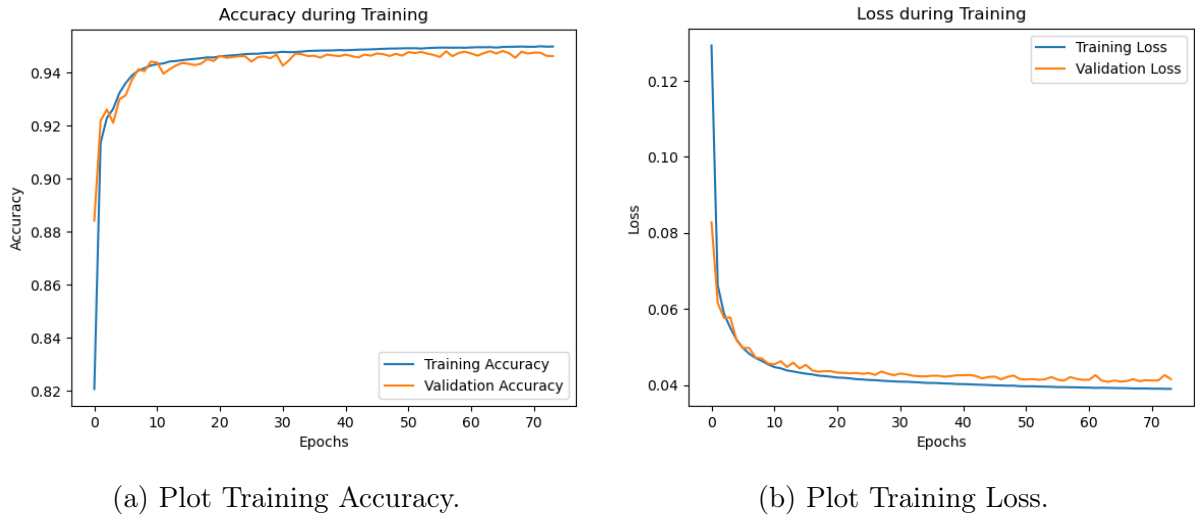


Figure 5.2: Plot of my Dataset with 5 Classes.

In the personal dataset with 5 classes, the addition of the *mqtt\_attack* class leads to a slightly slower convergence than in the case with 4 classes. The accuracy stabilizes around 95 per cent, with a final loss of approximately 0.04, which is higher than in the 4-class configuration. The addition of the *mqtt\_attack* class introduces additional

complexity that makes it more difficult for the model to achieve perfect separation between classes. However, the stability of the accuracy and loss curves indicates that the model still generalizes well, maintaining a very good classification capability even in the presence of an additional class.

### 5.2.3 Results of TON\_IoT Dataset

The results obtained in Table 5.3 with the TON\_IoT dataset show an **accuracy** on the test set of **92.34%** and a **loss** of **0.0584**. The other classification metrics (Accuracy, Recall and F1-score), for both classes, are close to 92%.

Class	Precision	Recall	F1-Score	Support
0	0.90	0.95	0.93	25000
1	0.95	0.90	0.92	25000
<b>Accuracy</b>			0.92	50000
<b>Macro Avg</b>			0.92	50000
<b>Weighted Avg</b>			0.92	50000

Table 5.3: Classification Report Dataset with 5 Classes.

Below in Figure 5.3 I will show the plot of accuracy and loss in the training phase.

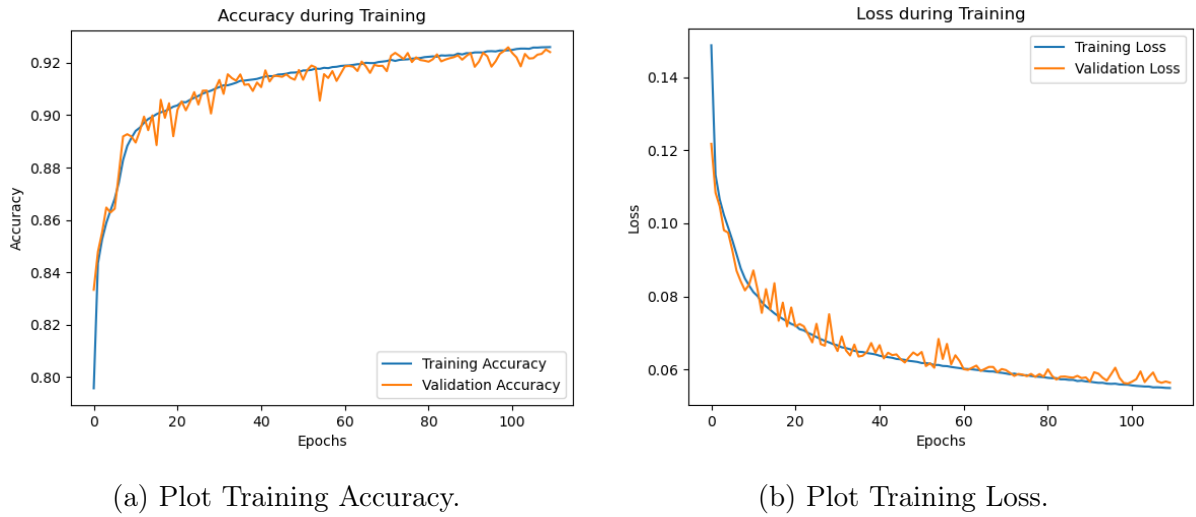


Figure 5.3: Plot of TON\_IoT Dataset with 4 Classes.

In the accuracy and loss graphs of the TON\_IoT dataset, an initial growth of the metrics is observed, but with a plateau that stabilizes at lower levels than in the other two datasets. The accuracy stabilizes around 92%, while the final loss remains higher, around 0.0584. This difference suggests that the TON\_IoT dataset is less representative of the key characteristics of the classes than the personal dataset, requiring more epochs to stabilize the values and with a lower overall performance.



### 5.2.4 Evaluating pre-trained Siamese Network

The results of the cross-tests between the models pre-trained on the created dataset and TON\_IoT show important differences in the generalization capabilities of the Siamese network. When the network is pre-trained on the customized dataset with 4 classes and tested on TON\_IoT (Table 5.4), the accuracy drops considerably, stabilizing at around **59.5%**, while the accuracy is slightly higher, at **64.66%**, if the network is trained on the customized dataset with 5 classes. In both cases, however, the model struggles to distinguish the classes in the TON\_IoT dataset. The high loss, together with the precision and recall remaining around 60-65%, indicates that the features present in the customised dataset are different and distinctive from those in the TON\_IoT dataset. This suggests a limited overlap in patterns between the two datasets, highlighting how TON\_IoT is less representative of the specific classes and more detailed features of the created dataset.

Class	Precision	Recall	F1-Score	Support
0	0.61	0.54	0.57	25000
1	0.59	0.65	0.61	25000
<b>Accuracy</b>			0.59	50000
<b>Macro Avg</b>	0.60	0.59	0.59	50000
<b>Weighted Avg</b>	0.60	0.59	0.59	50000

Class	Precision	Recall	F1-Score	Support
0	0.67	0.59	0.62	25000
1	0.63	0.71	0.67	25000
<b>Accuracy</b>			0.65	50000
<b>Macro Avg</b>	0.65	0.65	0.65	50000
<b>Weighted Avg</b>	0.65	0.65	0.65	50000

Table 5.4: Classification Report of Siamese pre-trained on my dataset.

When the network pre-trained on TON\_IoT is tested on the dataset created with 4 classes (Table 5.5), the accuracy is similar, **61.17%**, accompanied by a loss of 0.2980. Again, accuracy and recall are low, showing that the model trained on TON\_IoT fails to effectively capture the unique features of the created dataset. This is a clear indication that TON\_IoT does not contain the same distinctive features present in the created dataset, which is evidently more specific and better balanced to represent the relevant classes for an IoT architecture. The results, therefore, suggest that the created dataset is qualitatively superior.

Class	Precision	Recall	F1-Score	Support
0	0.60	0.65	0.63	25000
1	0.62	0.57	0.59	25000
<b>Accuracy</b>			0.61	50000
<b>Macro Avg</b>	0.61	0.61	0.61	50000
<b>Weighted Avg</b>	0.61	0.61	0.61	50000

Table 5.5: Classification Report of Siamese pre-trained on TON\_IoT.

### 5.2.5 Report of results obtained

The results show that the dataset created with 4 classes offers the best overall performance. The high accuracy, close to 99%, and low loss indicate that the model is highly effective in distinguishing the different classes present. The training graphs show rapid convergence and very low and stable loss, suggesting that the dataset is well balanced and representative of the distinctive features of the classes, facilitating effective learning.

The addition of the fifth class in the created dataset results in a decrease in accuracy to 95.02% and an increase in loss. This result indicates that the *mqtt\_attack* class introduces additional complexity into the classification problem that makes class separation more difficult. The training graphs of the 5-class dataset show slower convergence and a plateau at slightly lower levels, suggesting that the model requires a larger number of epochs to achieve maximum performance. However, the stability of the training and validation curves shows that the model continues to generalize well despite the added complexity.

The TON\_IoT dataset with 4 classes shows the lowest performance, with an accuracy of 92.34% and a loss that stabilizes at higher values than the two customized datasets. This result is evidenced by the graphs, which show slower convergence and a lower ability of the model to reduce loss, suggesting that the TON\_IoT dataset is less representative of the key characteristics of the classes, probably due to its smaller size or lower data quality.

In contrast, the results of cross-tests between the models pre-trained on the created dataset and on TON\_IoT further strengthen the evidence of the superiority of the created dataset. When the Siamese model trained on the created dataset is tested on TON\_IoT, it shows a high ability to distinguish classes, indicating that these are well defined and that distinctive information is clearly present in the data. In contrast, the model trained on TON\_IoT fails to recognize patterns present in the created dataset as well, confirming that TON\_IoT does not allow for effective generalization.

In conclusion, the comparative analysis suggests that the dataset created with 4 classes is superior to both the dataset with 5 classes and the TON\_IoT dataset. The high performance obtained indicates that the dataset created is of high quality, with well-defined classes and a sufficient volume of data to effectively train the model. This makes it a good candidate to be used by the scientific community as a benchmark in the study of attacks on IoT architectures.

### 5.3 Performance of Transfer Learning

I will now go on to show and analyze the results of the transfer learning that highlight the success of the training process, which utilized the created dataset as the basis for pre-training and subsequently used the TON\_IoT dataset for final refinement. The analysis of the training plots of accuracy and loss, combined with the classification metrics obtained, provides a clear overview of the model's ability to adapt and generalize to a more complex context.

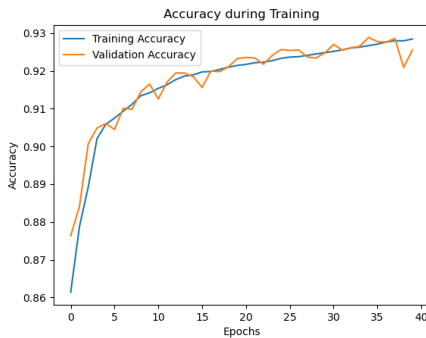
#### 5.3.1 Pre-Train on 4-Class Dataset and Fine-Tuning on TON\_IoT

The model pre-trained on the customized dataset with 4 classes and subsequently refined on TON\_IoT achieved an accuracy of **91.60%**. The accuracy and recall metrics remained high, with values ranging between 89% and 95% for both classes, and an F1-score of 92%, suggesting that the model was able to maintain a good discriminative ability even after the transition to the TON\_IoT dataset (Table 5.6).

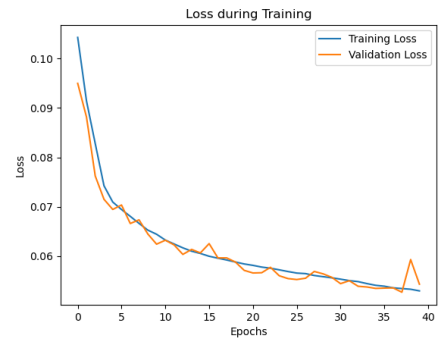
Class	Precision	Recall	F1-Score	Support
0	0.89	0.95	0.92	25000
1	0.95	0.88	0.91	25000
<b>Accuracy</b>			0.92	50000
<b>Macro Avg</b>	0.92	0.92	0.92	50000
<b>Weighted Avg</b>	0.92	0.92	0.92	50000

Table 5.6: Classification Report of Transfer Learning with my dataset of 4 classes.

The accuracy and loss graphs during training show a very positive trend. In the first training cycles, a rapid increase in accuracy is observed, followed by a stabilization around 93%. The closeness between the training and validation curves is a clear indicator that the model has not been overfitted and has maintained a good level of generalization on the validation data.



(a) Plot Training Accuracy.



(b) Plot Training Loss.

Figure 5.4: Plot of Transfer Learning with my dataset of 4 classes.

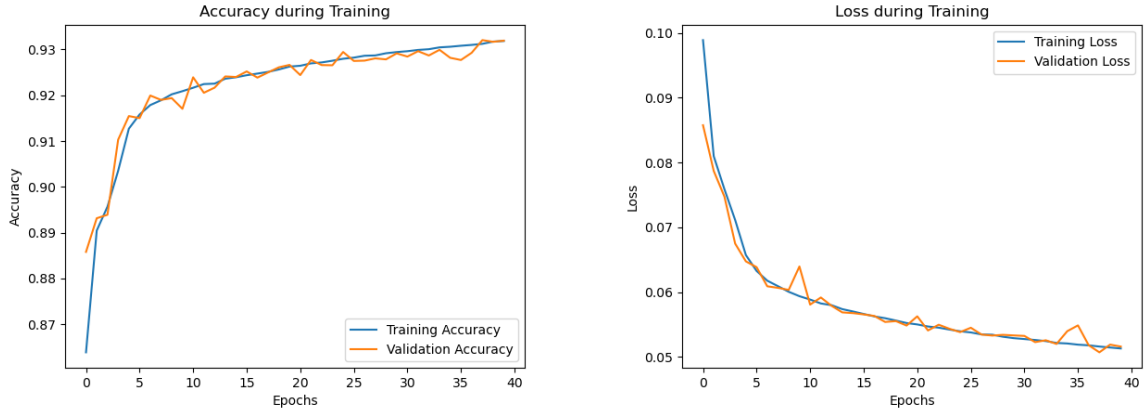
### 5.3.2 Pre-Train on 5-Class Dataset and Fine-Tuning on TON\_IoT

The model pre-trained on the dataset created with 5 classes achieved a slightly higher accuracy of **91.82%**, maintaining an F1-score of 92%. Accuracy and recall remained stable with similar high values to the model with 4 classes, suggesting that the additional class (mqtt\_attack) did not significantly penalize the model's ability to generalize to the TON\_IoT data (Table 5.7).

Class	Precision	Recall	F1-Score	Support
0	0.89	0.95	0.92	25000
1	0.95	0.89	0.92	25000
<b>Accuracy</b>			0.92	50000
<b>Macro Avg</b>	0.92	0.92	0.92	50000
<b>Weighted Avg</b>	0.92	0.92	0.92	50000

Table 5.7: Classification Report of Transfer Learning with my dataset of 5 classes.

In the plots of accuracy and loss, the accuracy grows rapidly in the first epochs and stabilizes at around 93% towards the end of training, with a slight fluctuation in validation accuracy. The validation loss follows the training curve closely, indicating a good capacity for generalization even in the presence of the increased complexity due to the fifth class. The model reduced the loss to very low values (around 0.05), suggesting that the learning process was effective.



(a) Plot Training Accuracy.

(b) Plot Training Loss.

Figure 5.5: Plot of Transfer Learning with my dataset of 5 classes.

### 5.3.3 Report of results obtained

The overall results show that both models achieved similar results, with accuracy around 91-93% and low losses. Despite the addition of the mqtt\_attack class, the model pre-trained on the dataset with 5 classes did not suffer a significant decrease in performance.

On the contrary, it maintained accuracy, precision and recall metrics comparable to the model trained with 4 classes, demonstrating good adaptability even in a more complex context.

Thus, the transfer learning process adopted demonstrated the effectiveness of robust pre-training followed by specific fine-tuning. The final results, characterized by high accuracy and good generalization, suggest that the created dataset and the resulting model can be a significant contribution to IoT security research, offering a robust and well-balanced benchmark for further studies and developments.

### 5.4 Comparative Analysis

The overall analysis of the results obtained from various classifiers, the Siamese network and transfer learning clearly demonstrates the high value of the dataset created, particularly in its version with 4 classes. The dataset performed exceptionally well, achieving extremely high accuracies (up to 99% with the Siamese network and Random Forest), suggesting that it is well balanced and representative of the distinctive features of the classes. This representativeness and the quality of the data collected allow for effective generalization and make the dataset an excellent candidate to be used as a benchmark for IoT architecture security research.

The addition of the *mqtt\_attack* class in the 5-class dataset, although it introduced more complexity, was a crucial choice to more realistically represent attack scenarios in the IoT context. The *mqtt\_attack* class represents a specific but increasingly common type of threat in IoT systems, since the MQTT protocol is one of the most widely used for communication between devices in distributed architectures. The results obtained suggest that although the addition of this class reduced the overall accuracy, the model still managed to distinguish classes well, highlighting its ability to deal with a greater variety of attacks. This ability to handle the *mqtt\_attack* class is essential to ensure accurate detection of real attacks in the IoT world, where the MQTT protocol plays a key role and represents a vulnerable attack vector.

Furthermore, the transfer learning approach used with the dataset created and subsequently refined with TON IoT demonstrated the effectiveness of starting with a high quality dataset and then adapting to broader contexts. Despite the increased complexity introduced by *mqtt\_attack*, the model maintained high performance, demonstrating that the inclusion of this class not only makes the dataset more realistic and representative of current IoT security challenges, but also allows the models to be better prepared to deal with more complex and realistic scenarios.

Therefore, the dataset created has not only proven to be qualitatively superior to TON\_IoT, but is also a valuable tool for the scientific community. The inclusion of the *mqtt\_attack* class represents a fundamental step towards the construction of a more complete bench-

mark suitable for representing the real risks of the IoT world. This makes the dataset created an indispensable resource for the research and development of advanced techniques for detecting attacks in IoT networks, offering a robust and well-balanced benchmark for the scientific community and for further studies in this rapidly evolving field.

# Chapter 6

## MCU-Deployed SNN for IDS

This chapter describes the process of deploying a Siamese neural network on a microcontroller, with a special focus on the ESP32 using TensorFlow Lite for model conversion. The use of microcontrollers such as the ESP32, which are characterized by limited resources in terms of both computing power and memory, requires specific techniques to ensure efficient deployment of neural networks, especially for real-time applications such as IDS in IoT environments.

The deployment of the Siamese network on ESP32 was made possible through the conversion of the original model into a TensorFlow Lite version, specially optimized for execution on low-resource hardware. Next, the tests performed to evaluate the effectiveness and performance of the implemented system are described. The tests concerned the model's ability to detect whether an attack is present or not, with the aim of verifying whether the model's performance on the system was equal to that obtained locally, without a significant drop in performance. In addition, a mathematical analysis of inference times was conducted to determine whether the microcontroller was able to handle the required operations in a time compatible with the needs of a real-time detection system. These evaluations were crucial in determining the feasibility of deploying the Siamese network in real-world scenarios, where the microcontroller's resource limitations can significantly affect the detection capability.

Finally, the problems encountered during the deployment and testing process are discussed. Among these, one of the main obstacles was the excessively long inference times, which compromised the speed of decision-making in real-time detection contexts. In addition, the model conversion presented numerous difficulties, partly related to the lack of reliability of the TensorFlow documentation, which made the adaptation process for the ESP32 low-resource environment complex. To address these issues, solutions are proposed to further improve performance, such as hardware-software optimizations and the possible integration of specific accelerators, which can significantly reduce inference times.

## 6.1 Embedded Hardware Selection

The selection of embedded hardware for the deployment of the Siamese neural network-based sensing model was a crucial step to ensure the effectiveness and sustainability of the system in real-world scenarios, especially in the context of IoT networks. After a thorough evaluation of the available options, the ESP32 microcontroller was selected, in particular the TTGO LoRa variant. This choice was motivated by several technical and practical factors, in particular the support offered by TensorFlow Lite for execution on resource-limited platforms.

The ESP32 TTGO LoRa is an advantageous solution for the implementation of intrusion detection systems (IDS) due to its hardware features, such as the amount of memory available: 16MB of Flash memory and 520KB of SRAM memory. In addition, TensorFlow Lite’s compatibility with the ESP32 architecture allows it to perform quantized pattern inference in an optimized way, making the most of the microcontroller’s limited computational capabilities.

The TTGO LoRa version of the ESP32 offers additional advantages over other variants of the same microcontroller, notably the presence of a LoRa module that enables long-distance communication with low power consumption. This feature is particularly useful in remote IoT scenarios, where the ability to communicate relevant data without relying on Wi-Fi or wired networks is crucial. The availability of sufficient memory and the ESP32’s dual-core architecture also enable efficient handling of model inference and network operations, while keeping response times low.

In summary, the choice of the ESP32 TTGO LoRa as the hardware platform for the deployment of the Siamese neural network was motivated by its compatibility with TensorFlow Lite, its connectivity capabilities and its flexibility, making it an ideal option for intrusion detection applications in IoT contexts. This platform provided a good compromise between performance, cost, energy efficiency and the ability to adapt to the specific requirements of distributed IoT networks.

## 6.2 Model conversion for Embedded Environments

The conversion of the model for the embedded environment represented a crucial phase to ensure compatibility and efficiency in executing the Siamese network model on a resource-constrained microcontroller, in this case, the ESP32 TTGO LoRa. To achieve this goal, TensorFlow Lite was used, which is specifically designed to enable the execution of machine learning models on embedded devices with limited computational and memory capabilities. [50].



The conversion process was carried out using the following Python code, which utilizes TensorFlow Lite features to transform the original model into a quantized version suitable for resource-constrained environments:

```
def load_and_convert_model(path):  
    # Load Siamese Model  
    siamese_model = (SiameseNet(input_shape=(31, 1, 1))).load_saved_model(path)  
  
    # Conversion Siamese Model  
    converter = tf.lite.TFLiteConverter.from_keras_model(siamese_model)  
    tflite_model = converter.convert()  
  
    # Save the TFLite model  
    with open('siamese_model.tflite', 'wb') as f:  
        f.write(tflite_model)
```

In the initial phase, the Siamese network model was loaded from the pre-trained version and then converted using TensorFlow Lite's `TFLiteConverter`. The converter allows a Keras model to be transformed into an optimized, lighter model and then saved in a `.tflite` file ready to be loaded and used on embedded devices. It is important to note that the model was not further optimized by quantitation (e.g. conversion of weights from float to int) as this process caused specific problems when running the model on ESP32 with Arduino code. In particular, during the allocation of tensors on the microcontroller, the change from float to int data caused incompatibilities that prevented the tensors from being initialized correctly. This problem, related to the ESP32's limited hardware support and the difficulty in handling integer representations for complex networks such as Siamese, led to the decision to keep the model in the original float32 format to ensure correct execution.

The second part of the code concerns the testing of the converted model, to verify that the conversion has taken place correctly and that the model is capable of making inferences. This step is important to ensure that the model has not lost crucial information or precision during the conversion process.

```
def test_tflite_model():  
    # Load the TFLite model  
    interpreter = tf.lite.Interpreter(model_path='siamese_model.tflite')  
    interpreter.allocate_tensors()  
  
    # Get input and output details  
    input_details = interpreter.get_input_details()  
    output_details = interpreter.get_output_details()  
  
    # Prepare sample input data  
    input_shape_left = input_details[0]['shape']  
    input_shape_right = input_details[1]['shape']  
    input_data_left = np.random.rand(*input_shape_left).astype(np.float32)  
    input_data_right = np.random.rand(*input_shape_right).astype(np.float32)
```

```
# Set the tensor
interpreter.set_tensor(input_details[0]['index'], input_data_left)
interpreter.set_tensor(input_details[1]['index'], input_data_right)

# Invoke the model
interpreter.invoke()

# Get the output
output_data = interpreter.get_tensor(output_details[0]['index'])
print("Output:", output_data)
```

## 6.3 Testing and Validation on Embedded Hardware

After the conversion of the Siamese model into TensorFlow Lite format, it was necessary to verify its correct functioning on an embedded device. For this reason, code was developed on the Arduino to test the implementation and measure the performance of the converted Siamese neural network model by performing inference directly on the ESP32 TTGO LoRa. This step is crucial to ensure that the model functions correctly even under the resource-limited operating conditions typical of IoT microcontrollers.

For testing purposes, a file called `test_pairs.h` was created, containing 100 input pairs to be used as a test set for the Siamese network. The objective was to assess whether the model could correctly distinguish between similar and non-similar input pairs by calculating the similarity for each pair. The Arduino code is mainly divided into two sections: the setup and the loop.

- **Setup: initializing the model**

In the setup, the model is prepared for use by ESP32. The `setup_model()` function is designed to load the TFLite model, allocate the necessary tensors and prepare the interpreter for inference:

```
void setup_model() {
    // Load Model
    model = tflite::GetModel(g_model);
    if (model->version() != TFLITE_SCHEMA_VERSION) {
        MicroPrintf(
            "Model provided is schema version %d not equal to supported "
            "version %d.",
            model->version(), TFLITE_SCHEMA_VERSION
        );
        return;
    }

    // Pull in only the operation implementations we need.
    static tflite::MicroMutableOpResolver<9> resolver;
    resolver.AddConv2D();
    resolver.AddRelu();
    resolver.AddLogistic();
    resolver.AddMaxPool2D();
}
```

```
resolver.AddReshape();

resolver.AddSquaredDifference();
resolver.AddSum();
resolver.AddMaximum();
resolver.AddSqrt();

// Build an interpreter to run the model with.
static tf::MicroInterpreter static_interpreter(model, resolver, tensor_arena,
↪ kTensorArenaSize);
interpreter = &static_interpreter;

// Allocate memory from the tensor_arena for the model's tensors.
TfLiteStatus allocate_status = interpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
    MicroPrintf("AllocateTensors() failed");
    return;
}

// Obtain pointers to the model's input and output tensors.
input_a = interpreter->input(0);
input_b = interpreter->input(1);
output = interpreter->output(0);
}
```

This function is responsible for initializing the model, resolving the operations required for the model (such as Conv2D, Relu, Logistic, etc.). After loading the model, tensors are allocated, and pointers to the inputs and outputs are obtained to prepare them for inference. The presence of `MicroMutableOpResolver` allows only those operations necessary for the model to be included, reducing the computational load and optimizing the use of limited memory.

- **Loop: inference and time analysis**

In the loop, model inferences are made using the input pairs stored in the `test_pairs.h` file. The function `load_inputs(int pair_index)` takes care of loading the input pairs for each inference, ensuring that the correct data are assigned to the network inputs:

```
void load_inputs(int pair_index) {
    if (pair_index >= NUM_PAIRS) {
        Serial.println("Indice coppia fuori range!");
        return;
    }

    for (int i = 0; i < FEATURE_SIZE; i++) {
        input_a->data.f[i] = pairs_a[pair_index][i][0][0];
        input_b->data.f[i] = pairs_b[pair_index][i][0][0];
    }
}
```

After loading the data, the `run_inference()` function is used to perform the inference with the TensorFlow Lite interpreter:

```
float run_inference() {
    TfLiteStatus invoke_status = interpreter->Invoke();
    if (invoke_status != kTfLiteOk) {
        Serial.println("Inferenza fallita!");
        return -1.0f;
    }

    float similarity = output->data.f[0];
    return similarity;
}
```

Inference is then performed and the similarity calculated by the model is used to determine whether the two input pairs are similar or not. In this case, the obtained similarity value is compared with a threshold (0.5) to determine whether the pairs belong to the same class.

During each loop cycle, detailed statistical metrics are also collected to evaluate the performance of the model on the microcontroller. Metrics include:

- **Average Time Per Prediction:** the average time taken to execute each prediction.
- **Standard Deviation of Time:** the variation of prediction times from the average.
- **Minimum and Maximum Time per Prediction:** the lower and upper limits of the prediction times.

These metrics are calculated for each step in the process: input loading, model inference and post-processing of results (including checking the correctness of the prediction). This information is essential to assess the efficiency of the model deployment on the ESP32 and verify whether the performance is adequate for real-time use. During the loop, the results of each prediction are collected and printed out, showing the calculated similarity, the predicted value and the actual value. At the end of the test loop, overall statistics such as overall accuracy and time metrics for each step are also calculated and printed.

The objective of these tests is to ensure that the converted model can actually be run on an ESP32 microcontroller with acceptable performance, while maintaining sufficiently low inference times and high accuracy. In addition, these metrics help identify possible bottlenecks and evaluate possible hardware or software optimizations needed to further improve the performance of the microcontroller-based IDS system.

## 6.4 Results

The results of the tests performed (Table 6.1) on the ESP32 TTGO LoRa microcontroller show an accuracy of **99.00%**, perfectly in line with that obtained during local testing. This is extremely positive, as it indicates that the Siamese network model, despite the complexity of the embedded context and the hardware limitations of the ESP32, was able to maintain its generalization and classification capability without any loss of performance compared to the development environment.

A detailed analysis of the execution times of the various operations shows an average time per prediction of approximately 7.50 seconds, with a standard deviation of 0.00 seconds, suggesting a very stable behaviour of the system in terms of inference duration. The minimum and maximum times for each prediction are between 7.5027 seconds and 7.5040 seconds, values that are very close to each other, testifying to the consistency and predictability of the model's performance.

The average input loading time is extremely low at 0.00002908 seconds, with a very low standard deviation. This indicates that the input preparation phase, i.e. the assignment of input data to model tensors, is extremely efficient and does not represent a significant bottleneck in the entire inference process. The minimum and maximum times for input loading are very close, varying between 0.00002800 seconds and 0.00003400 seconds, further confirming the stability of this phase.

The inference phase of the model is, understandably, the most time-consuming part, with an average inference time of 7.5035 seconds. Here too, the standard deviation is practically zero, suggesting that the inference time is constant and predictable. This finding is particularly important as it indicates that, despite the hardware limitations of the ESP32, the model manages to perform inference reliably and without significant variations in execution time.

Finally, the post-processing phase showed an extremely low average time of 0.00000673 seconds, with an insignificant standard deviation. This confirms that the final part of the inference process, which includes verifying the prediction and collecting the results, has a minimal impact on the overall performance in terms of time.

In general, the test showed that the entire inference process can be performed in between **7.5027 seconds** and **7.5040 seconds**, values that indicate consistent and predictable behaviour. However, it is important to emphasise that these high inference times are not adequate for real-time anomaly detection applications, where the ability to respond to suspicious events in a few milliseconds is crucial to secure IoT architectures. In real-time intrusion detection scenarios, these times would not allow timely reaction to attacks, increasing the risk of exposure.

Instead, these values are more suitable for scenarios in which network traffic analysis can be carried out at a later stage, e.g. for a retrospective study of attacks or for security audit activities. In these cases, the model offers very high accuracy, maintaining performance similar to that obtained locally, and is a useful tool for in-depth analysis of network traffic and accurate classification of detected activities. This makes the model implemented on ESP32 a viable option for applications that do not require real-time responses but, rather, accurate detection and post-event analysis of network traffic in IoT contexts.

Metric	Avg	Std Dev	Min	Max
Acc (%)	99.00	-	-	-
Pred Time (s)	7.5036	0.0000	7.5027	7.5040
Load Time (s)	0.000029	0.000001	0.000028	0.000034
Infer Time (s)	7.5035	0.0000	7.5027	7.5040
Post Time (s)	0.000007	0.000000	0.000006	0.000007

Table 6.1: ESP32 Test Results for Accuracy and Inference Times.

## 6.5 Problems and Solutions

During the process of deploying the Siamese model on the ESP32, various technical problems were encountered, mainly related to the model’s compatibility with the embedded environment and the microcontroller’s limitations in terms of computing resources and memory. The main problems encountered and the possible solutions proposed to deal with them are described in detail below.

1. **TensorFlow documentation not in line with the code:** one of the main problems encountered during the model conversion process was the discrepancy between the official TensorFlow Lite documentation and the actual code. Often, the instructions in the documentation were not up-to-date or did not correspond to the latest version of the library, which led to difficulties in following the suggested steps for model conversion. The lack of clarity in the documentation caused an increase in development time and several debugging attempts to resolve unexpected errors.

**Proposed Solution:** a possible solution would be to create a more specific practical guide, dedicated to the most commonly used models in microcontrollers, containing detailed and up-to-date examples. In addition, one could contribute to the TensorFlow community with issue reports and pull requests to improve the official documentation.

2. **Model conversion from Conv1D to Conv2D:** another significant problem was the need to convert the model from Conv1D to Conv2D, along with the conversion of the corresponding Max Pooling layers. This change was necessary because TensorFlow Lite does not directly support Conv1D operations, making it impossible to directly convert the model as it was initially trained. The need to adapt the model resulted in a structural change to the network, requiring new tests to verify that performance was not compromised.

**Proposed Solution:** one solution to reduce the difficulty of layer conversion would be to develop a dedicated tool for automatic model conversion, capable of translating unsupported layers (e.g. Conv1D) into alternative layers (e.g. Conv2D) while maintaining the original characteristics of the model. This tool could automatically identify non-compatible layers and suggest acceptable alternatives, saving time and effort during the implementation phase.

3. **Excessively high model inference time:** a major problem for deployment on a microcontroller such as the ESP32 was the excessively high inference time of around 7.5 seconds per prediction. This latency is too high for scenarios requiring real-time anomaly detection, limiting the model’s applicability for rapid and timely threat detection. Such long inference times compromise the immediate response capability of the system, making it unsuitable for critical applications such as real-time protection of IoT networks.

**Proposed Solutions:**

- *Model Pruning:* an effective solution to reduce inference times could be the application of pruning techniques, which consist of reducing the complexity of the model by removing non-essential weights. This process could reduce the number of parameters and memory consumption, improving inference times. However, special care must be taken during quantisation and layer conversion, as some optimisations (such as converting from float32 to int8) may lead to compatibility problems, as highlighted during initial testing.
- *Reducing the dimensionality of input data:* another possible solution is the reduction of input dimensionality. Through preprocessing techniques, only the most relevant features of the data can be extracted, reducing the computational load of the model. This would allow the accuracy of the model to be maintained while improving the speed of inference.
- *Integration of hardware accelerators:* The use of hardware accelerators such as TPUs (Tensor Processing Units) could significantly improve inference times. Although ESP32 does not directly support TPUs, there are microcontroller

versions that offer integrated hardware accelerators for machine learning operations. One possible improvement could be to consider using more advanced microcontrollers that are compatible with these accelerators to reduce inference times.

- *Inference distribution*: another possible solution could be to distribute the inference load over several devices or to perform part of the inference on an edge server. In this way, the microcontroller could take care of the preprocessing operations and only the final decisions, while the computationally more time-consuming part could be handled by a device with greater capabilities.

Therefore, the problems encountered during the microcontroller deployment process, such as layer compatibility limitations and excessive inference times, highlight the need to find practical solutions for adapting complex models to embedded environments. The proposed solutions, including the development of dedicated conversion tools, model optimisation through pruning and evaluation of new hardware, can help to significantly improve the feasibility of deploying machine learning models on microcontrollers, making them more suitable for real-time anomaly detection scenarios in IoT networks.



# Chapter 7

## Conclusions and Future Developments

This chapter concludes the research work carried out, summarising the main results obtained and identifying the limitations of the study. Furthermore, guidelines for future developments are proposed in order to improve the effectiveness of the proposed system and further contribute to research in the field of IoT security.

### 7.1 Summary of Obtained Results

The research presented in this thesis has shown extremely promising results, with particular reference to the quality of the dataset developed and the effectiveness of the Siamese network for anomaly detection in IoT environments.

The dataset created is of particular importance, as it includes MQTT traffic, a crucial protocol for IoT communications. Unlike many other datasets in the literature, the dataset developed was built by collecting data from real, not simulated IoT devices, giving it superior representativeness and reliability for application in real scenarios. This characteristic makes it particularly suitable for understanding and detecting attacks that exploit vulnerabilities in IoT devices, representing a significant contribution to the research community. The inclusion of MQTT traffic - crucial for communication between IoT devices - provides a rare level of detail that is crucial for building detection models that are able to identify attacks involving this specific protocol, which is becoming increasingly popular and vulnerable. The performance of the dataset, which has reached very high levels of accuracy, shows that it is well balanced, rich in meaningful information, and representative of the various types of traffic that characterise modern IoT networks. This makes it an excellent candidate for use in scientific research, especially as a benchmark for evaluating new intrusion detection algorithms.

The Siamese network used in this study proved particularly effective in distinguishing between different classes of network traffic, acting as an Intrusion Detection System (IDS). The model's ability to maintain high accuracy, even after being deployed on an embedded device such as the ESP32, highlights its adaptability and practicality for resource-limited

environments. This deployment demonstrates that Siamese neural networks are a viable solution for lightweight anomaly detection in embedded IoT systems, especially in the presence of critical protocols such as MQTT, where malicious activity needs to be detected with minimal resource usage.

Experiments deploying the model on an ESP32 microcontroller using TensorFlow Lite have shown that, despite the challenges posed by resource limitations, it is possible to maintain model performance similar to that achieved locally. This represents a significant advance for the use of machine learning models directly on IoT devices, enabling a new generation of autonomous and intelligent devices capable of detecting real-time attacks in real-world environments.

### 7.2 Study Limitations

Despite the promising results, some limitations have to be taken into consideration to further improve the applicability of the Siamese network and the developed dataset in IoT security. The main limitation that emerged was the high inference time when the Siamese model was deployed on the ESP32. The inference time, which was around 7.5 seconds per prediction, is a critical issue for real-time anomaly detection, introducing excessive latency in applications that require an immediate response to attacks.

Another critical issue was compatibility problems during model conversion. The need to convert Conv1D layers to Conv2D, along with the corresponding pooling layers, due to the lack of compatibility of TensorFlow Lite with Conv1D operations, introduced additional complexities. Although the conversion process was successfully completed, it required structural adjustments and a testing phase to ensure that performance was not compromised.

Finally, the dataset also has some limitations. Although it proved to be highly representative and suitable for identifying several common attacks, it is still limited to a specific set of devices and attack types. Expanding the dataset to include more IoT devices and additional attacks would make it even more versatile and applicable to a wider range of real-world scenarios.

### 7.3 Proposals for Future Research

The promising results obtained suggest several directions for future research that could further improve both dataset and anomaly detection capabilities in IoT environments.

1. **Expanding the Dataset:** one of the main opportunities for future research lies in expanding the dataset. The current dataset can be improved by including data from other real IoT devices and integrating new services commonly used in IoT networks. The addition of new devices and services would generate additional attack

vectors and, consequently, new attack classes. Such an expansion would significantly increase the representativeness of the dataset, making it more suitable for use in diverse IoT environments and providing a more robust benchmark for the scientific community. The use of real devices, rather than simulators, would continue to provide a highly reliable and realistic dataset, making it particularly valuable for the security assessment of IoT networks.

2. **Development of a model conversion tool:** another potential line of research concerns the development of a tool to simplify the deployment of machine learning models on embedded hardware. This tool could be designed to convert Python models directly into C code, making them suitable for microcontrollers, thus enabling simple and direct integration of the model on a wide range of embedded devices, not just a limited subset. Such a tool could solve many of the compatibility issues encountered, making the deployment process faster and smoother.
3. **Optimizing the model for embedded environments:** given the high inference time observed, future research should focus on techniques to optimise the model for the embedded environment. Techniques such as pruning, quantisation or knowledge distillation could be applied to reduce the size and complexity of the model without significantly compromising its accuracy. These optimisations could help reduce inference times and make the model more suitable for real-time applications. Furthermore, an interesting direction could be to test the possibility of training the model directly on embedded devices with scarce computational resources, thus making the devices capable of learning dynamically from new data.
4. **Integration of hardware accelerators:** another aspect to be explored is the integration of hardware accelerators or the use of more advanced microcontrollers offering native support for machine learning operations. The use of microcontrollers with integrated TPUs (Tensor Processing Units) could significantly reduce inference times, making the system suitable for real-time applications. A comparative analysis of different hardware platforms could help identify the best trade-off between cost, performance and power consumption, further enhancing the deployment of IDS in IoT environments.
5. **Real-time training and adaptation:** finally, a future direction could be to implement real-time training and adaptation capability on embedded systems. This capability would allow IoT devices to dynamically adapt to new observed data, creating a system that can not only detect known attack patterns but also learn new ones in real time. Such an approach would greatly increase the robustness of IoT networks, especially in environments where attack patterns evolve rapidly and static patterns may become obsolete.

## 7. CONCLUSIONS AND FUTURE DEVELOPMENTS

---

In conclusion, this study has laid the foundation for the use of machine learning-based IDS in resource-constrained IoT environments by demonstrating the effectiveness of the customised dataset and the Siamese network. The inclusion of MQTT traffic and the collection of data from real IoT devices give the dataset a unique value, making it particularly suitable for use as a benchmark in the scientific community. In the future, expanding the dataset, developing specialised tools, optimising models, improving hardware and adapting in real time are all directions that could significantly improve the applicability and impact of this research in the field of IoT security.

# Bibliography

- [1] S. Sicari, A. Rizzardi, L.A. Grieco, A. Coen-Porisini, Security, privacy and trust in Internet of Things: The road ahead, *Computer Networks*, Volume 76, 2015, Pages 146-164. [CrossRef]
- [2] C. Koliass, G. Kambourakis, A. Stavrou and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," in *Computer*, vol. 50, no. 7, pp. 80-84, 2017. [CrossRef]
- [3] ITU-T Recommendation Y.2060 - Overview of the Internet of Things. International Telecommunication Union, June 2012. [CrossRef]
- [4] Smith-Ditizio, A. A., & Smith, A. D. (2019). Using rfid and barcode technologies to improve operations efficiency within the supply chain. *Advanced Methodologies and Technologies in Business Operations and Management*, 1277-1288.[CrossRef]
- [5] Hada, H., & Mitsugi, J. (2011). Epc based internet of things architecture. *Proceedings of 2011 IEEE International Conference on RFID-Technologies and Applications (RFID-TA)* (pp. 527-532).[CrossRef]
- [6] Overview on Internet of Things (IoT) Architectures, Enabling Technologies and Challenges. [CrossRef]
- [7] J. Borghoff, "PRINCE—A low-latency block cipher for pervasive computing applications," in *Advances in Cryptology—ASIACRYPT*, X. Wang and K. Sako, Eds. Berlin, Germany: Springer, 2012, pp. 208–225.[CrossRef]
- [8] J. Guo, T. Peyrin, and A. Poschmann, "The PHOTON family of lightweight hash functions," in *Advances in Cryptology—CRYPTO*, P. Rogaway, Ed. Berlin, Germany: Springer, 2011, pp. 222–239.[CrossRef]
- [9] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, "SPONGENT: A lightweight hash function," in *Cryptographic Hardware and Embedded Systems—CHES*, B. Preneel and T. Takagi, Eds. Berlin, Germany: Springer, 2011, pp. 312–325.[CrossRef]
- [10] Information Technology Laboratory. (2019) Lightweight Cryptography Project. Accessed: Jun. 2019.[CrossRef]

- [11] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proc. IEEE*, vol. 102, no. 8, pp. 1126–1141, Aug. 2014.[CrossRef]
- [12] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proc. 44th ACM/IEEE Design Autom. Conf.*, San Diego, CA, USA, Jun. 2007, pp. 9–14.[CrossRef]
- [13] F. Meneghello, M. Calore, D. Zucchetto, M. Polese and A. Zanella, "IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices," in *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8182-8201, Oct. 2019.[CrossRef]
- [14] Sha'ari, Alya & Abdullah, Zubaile. (2022). A Comparative Study between Machine Learning and Deep Learning Algorithm for Network Intrusion Detection. *Journal of Soft Computing and Data Mining*. 3. 10.30880/jscdm.2022.03.02.005.[CrossRef]
- [15] Alyazia Aldhaheeri, Fatima Alwahedi, Mohamed Amine Ferrag, Ammar Battah, Deep learning for cyber threat detection in IoT networks: A review, *Internet of Things and Cyber-Physical Systems*, Volume 4, 2024, Pages 110-128, ISSN 2667-3452.[CrossRef]
- [16] M. Tavallaei, E. Bagheri, W. Lu, A.A. Ghorbani, A detailed analysis of the KDD CUP 99 data set, in: *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. Plus 0.5em Minus 0.4emIeee, 2009, pp. 1–6.[CrossRef]
- [17] N. Moustafa, J. Slay, Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set), in: *2015 Military Communications and Information Systems Conference (MilCIS)*. Plus 0.5em Minus 0.4emIEEE, 2015, pp. 1–6.[CrossRef]
- [18] I. Sharafaldin, A.H. Lashkari, A.A. Ghorbani, Toward generating A new intrusion detection dataset and intrusion traffic characterization, *ICISSp 1* (2018) 108–116[Cross-Ref]
- [19] N. Koroniotis, N. Moustafa, E. Sitnikova, B. Turnbull, Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: bot-iot dataset, *Future Generat. Comput. Syst.* 100 (2019) 779–796.[CrossRef]
- [20] M.-O. Pahl, F.-X. Aubet, All eyes on you: distributed multi-dimensional IoT microservice anomaly detection, in: *2018 14th International Conference on Network and Service Management (CNSM)*. Plus 0.5em Minus 0.4emIEEE, 2018, pp. 72–80.[Cross-Ref]
- [21] I. Cse-Cic-Ids2018, Cse-cic-ids2018 Dataset, 2022.[CrossRef]

- [22] I. Sharafaldin, A.H. Lashkari, S. Hakak, A.A. Ghorbani, Developing realistic distributed denial of service (DDoS) attack dataset and taxonomy, in: 2019 International Carnahan Conference on Security Technology (ICCST). Plus 0.5em Minus 0.4emIEEE, 2019, pp. 1–8.[CrossRef]
- [23] A. Hamza, H.H. Gharakheili, T.A. Benson, V. Sivaraman, Detecting volumetric attacks on iot devices via sdn-based monitoring of mud activity, in: Proceedings of the 2019 ACM Symposium on SDN Research, 2019, pp. 36–48.[CrossRef]
- [24] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood, A. Anwar, TON\_IoT telemetry dataset: a new generation dataset of IoT and IIoT for data-driven intrusion detection systems, IEEE Access 8 (2020), 165 130–165 150.[CrossRef]
- [25] I. Stratosphere Laboratory, Iot-23 Dataset, 2022.[CrossRef]
- [26] H. Hindy, E. Bayne, M. Bures, R. Atkinson, C. Tachtatzis, X. Bellekens, Machine learning based IoT intrusion detection system: an MQTT case study (MQTT-IoT-IDS2020 dataset), in: Selected Papers from the 12th International Networking Conference: INC 2020. Plus 0.5em Minus 0, 4emSpringer, 2021, pp. 73–84.[CrossRef]
- [27] M.A. Ferrag, O. Friha, D. Hamouda, L. Maglaras, H. Janicke, Edge-IIoTset: a new comprehensive realistic cyber security dataset of IoT and IIoT applications for centralized and federated learning, IEEE Access 10 (2022), 40 281–40 306.[CrossRef]
- [28] M. David Wagner and Drew Dean. Intrusion detection via static analysis. In Proceedings S&P'01, USA, 2001. IEEE Computer Society.
- [29] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In Proceedings of the 11th NDSS Symposium, 2004.
- [30] F. M. Tabrizi and K. Pattabiraman, "Flexible Intrusion Detection Systems for Memory-Constrained Embedded Systems," 2015 11th European Dependable Computing Conference (EDCC), Paris, France, 2015, pp. 1-12, doi: 10.1109/EDCC.2015.17. [CrossRef]
- [31] Esp32 Datasheet.[CrossRef]
- [32] WeMos D1 ESP8266 WiFi Board Datasheet.[CrossRef]
- [33] Arduino Uno WiFi Rev2 Board Datasheet.[CrossRef]
- [34] Raspberry Pi 3 Model B Board Datasheet.[CrossRef]
- [35] Buzzer 3 Pins Datasheet.[CrossRef]
- [36] Adafruit Fingerprint Sensor Datasheet.[CrossRef]

- [37] HC-SR501 Pir Motion Detector Datasheet.[CrossRef]
- [38] Raspberry Pi OS installation guide.[CrossRef]
- [39] hping3 documentation.[CrossRef]
- [40] Hydra documentation.[CrossRef]
- [41] MQTT-malaria GitHub.[CrossRef]
- [42] MQTTSA GitHub.[CrossRef]
- [43] Nmap documentation.[CrossRef]
- [44] Nessus documentation.[CrossRef]
- [45] Unicornscan documentation.[CrossRef]
- [46] Alsaedi, Abdullah & Moustafa, Nour & Tari, Zahir & Mahmood, Abdun & Anwar, Adnan. (2020). TON\_IoT Telemetry Dataset: A New Generation Dataset of IoT and IIoT for Data-Driven Intrusion Detection Systems. IEEE Access. 8. 10.1109/ACCESS.2020.3022862.[CrossRef]
- [47] Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., & Shah, R. (1993). Signature verification using a "Siamese" time delay neural network. In Advances in Neural Information Processing Systems (pp. 737-744). [CrossRef]
- [48] Chopra, S., Hadsell, R., & LeCun, Y. (2005). Learning a Similarity Metric Discriminatively, with Application to Face Verification. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR) (Vol. 1, pp. 539-546). [CrossRef]
- [49] Koch, G., Zemel, R., & Salakhutdinov, R. (2015). Siamese Neural Networks for One-shot Image Recognition. In Proceedings of the 7th International Conference on Learning Representations (ICLR). [CrossRef]
- [50] Tensorflow Lite Documentation. [CrossRef]



# List of Figures

1.1	SOA based IoT architecture. . . . .	7
1.2	Cloud based IoT architecture. . . . .	9
1.3	Cloud/Fog based IoT architecture. . . . .	10
1.4	IDS Classification. . . . .	17
1.5	Machine Learning methods. . . . .	18
1.6	Deep Learning methods. . . . .	18
2.1	My Cloud-based IoT Architecture. . . . .	32
2.2	Communication logic. . . . .	44
3.1	Traffic Sniffing Architecture. . . . .	54
5.1	Plot of my Dataset with 4 Classes. . . . .	89
5.2	Plot of my Dataset with 5 Classes. . . . .	90
5.3	Plot of TON_IoT Dataset with 4 Classes. . . . .	91
5.4	Plot of Transfer Learning with my dataset of 4 classes. . . . .	94
5.5	Plot of Transfer Learning with my dataset of 5 classes. . . . .	95