Desenvolvimento e Implementação de um Sistema de Deteção de Intrusões para Redes LoRaWAN

CARLOS MOISÉS MIRANDA TAVARES
(Licenciado)

Trabalho de Projeto para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientador:

Doutor Nuno Miguel Machado Cruz

Júri:
Presidente: Doutor Nuno Miguel Soares Datia
Vogais:
Doutor José Manuel de Campos Lages Garcia Simão
Doutor Nuno Miguel Machado Cruz

Novembro de 2025

Desenvolvimento e Implementação de um Sistema de Deteção de Intrusões para Redes LoRaWAN

CARLOS MOISÉS MIRANDA TAVARES
(Licenciado)

Trabalho de Projeto para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientador:

Doutor Nuno Miguel Machado Cruz

Júri:
Presidente: Doutor Nuno Miguel Soares Datia
Vogais:
Doutor José Manuel de Campos Lages Garcia Simão
Doutor Nuno Miguel Machado Cruz

Novembro de 2025

Agradecimentos

Quero agradecer ao meu orientador Nuno Cruz por me ter guiado desde o início do projeto para fazer o melhor trabalho possível e ter os melhores resultados, mantendo o otimismo e positividade bem como a sua disponibilidade em esclarecer-me todas as minhas dúvidas ao longo do projeto sempre que foi necessário.

Também devo agradecer à equipa da empresa Solvit por me ter fornecido o material necessário para fazer os testes à minha implementação, conforme necessário.

Agradeço ainda à minha família, amigos e namorada por todo o apoio que me deram desde o início da minha jornada escolar e académica, tanto nos momentos bons como nos mais difíceis. Também quero homenagear os meus avós que tiveram um papel muito importante na minha vida e sem dúvida me inspiraram a ser quem eu sou hoje.

E devo ainda agradecer a Deus, que me tem dado tudo que eu preciso para ser feliz e alcançar os meus objetivos em todos os níveis, tais como a nível académico e a nível pessoal.

Statement of integrity

I declare that this project work report is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

____

Lisbon, November 21st, 2025

Desenvolvimento e Avaliação de um Sistema de Deteção de Intrusões para Redes LoRaWAN

Resumo

A Internet das Coisas (IoT) tem revolucionado o setor tecnológico em diversas aplicações, tais como cidades inteligentes, medidores de temperatura e casas inteligentes, permitindo a objetos enviar informação *online* em tempo real. Long-Range Wide Area Networks (LoRaWAN) é um protocolo IoT que se destaca pela sua capacidade de comunicação a longa distância mantendo um baixo consumo de energia, através da sua arquitetura de rede aberta e distribuída. No entanto, apesar de este protocolo estar a ser cada vez mais adotado por organizações, empresas e outras entidades, o mesmo apresenta vulnerabilidades de segurança que, apesar de muitas vezes negligenciadas, podem ser exploradas por meio de ataques informáticos que comprometem os serviços fornecidos por quem usufrui da tecnologia ao violar a privacidade e integridade dos dados.

Com isso, foi proposto o desenvolvimento e implementação de um sistema de deteção de intrusões para redes LoRaWAN, que analisa o tráfego de rede LoRaWAN e classifica os pacotes como apresentando um comportamento anómalo ou não anómalo, com base nos padrões de comportamento aprendidos por algoritmos de Machine Learning implementados. Através de um estudo inicial do protocolo LoRaWAN e dos seus mecanismos de segurança, chegou-se à conclusão de que, apesar desses mecanismos protegerem o protocolo de certos ataques, os mesmos não são capazes de o proteger de outros ataques que apresentam um grande risco para a segurança do protocolo, tais como Jamming. Também foi efetuado um estudo teórico de algoritmos de *Machine Learning* e *Deep Learning* adequados para detetar ataques não resolvidos pelo protocolo LoRaWAN. O sistema de deteção de intrusões foi avaliado usando um conjunto de dados de pacotes LoRaWAN de um gateway de Lisboa (2020), com anomalias sintéticas injetadas para teste. Resultados mostraram que Histogram-Based Outlier Score alcançou os melhores resultados, embora a validação em ataques reais permaneça como trabalho futuro.

Palavras-chave: IoT, LoRaWAN, Sistema de Deteção de Intrusões, Machine Learning, Deep Learning

Development and Implementation of an Intrusion Detection System for LoRaWAN Networks

Abstract

Internet of Things (IoT) has been innovating the technology sector in applications such as smart cities, temperature gauges and smart homes, allowing objects to send information online in real time. Long-Range Wide Area Networks (LoRaWAN) is an IoT protocol that stands out for its ability to communicate over long distances while maintaining low energy consumption, through its open and distributed network architecture. However, even though this protocol is being increasingly more adopted by organizations, companies and other entities, it presents several security vulnerabilities which, despite being many times neglected, can be exploited through cyberattacks that disrupt the services provided by those who use the technology, violating data privacy and integrity.

Considering these aspects, it was proposed the development and implementation of an intrusion detection system for LoRaWAN networks, which analyses LoRaWAN network traffic and classifies packets as either presenting an abnormal or normal behaviour, based on behavioural patterns learnt by implemented Machine Learning algorithms. An initial study of LoRaWAN and its security mechanisms showed that, although these mechanisms protect the protocol from certain attacks, these mechanisms can't protect the protocol from another attacks that present a great risk to the protocol security, such as Jamming. A theoretical study of Machine Learning and Deep Learning algorithms that can be adequate for detecting attacks not resolved by LoRaWAN also took place. The intrusion detection system was evaluated using a dataset of LoRaWAN packets from a Lisbon gateway (2020), with synthetic anomalies injected for testing. Results showed that Histogram-Based Outlier Score achieved best performance, though validation on real attacks remains future work.

Keywords: IoT, LoRaWAN, Intrusion Detection Systems, Machine Learning, Deep Learning

List of Symbols and Acronyms

Roman alphabet

log	logarithm
W_d	hamming distance
$D(p\|\|q)$	kullback-leibler distance given probability mass functions P and Q
max	the highest number
A_i	value of bit at position ‘i’ on vector A
B_i	value of bit at position ‘i’ on vector B
$P(x)$	root of the baseline distribution on kullback-leibler divergence
$Q(x)$	learnt distribution on kullback-leibler divergence

Greek alphabet

$\sum$	sum of all numbers on a range
δ	Kronecker delta

Acronyms

AE	Autoencoder
AI	Artificial Intelligence
AIDS	Anomaly-Based Intrusion Detection System
API	Application Programming Interface
AppKey	Application Key
AppSKey	Application Session Key
AS	Application Server
BW	Bandwidth
CRC	Cyclic Redundancy Check
DBSCAN	Density-based spatial clustering of applications with noise
DevEUI	Device End-User Identifier
DDoS	Distributed Denial of Service
DNN	Deep Neural Networks
DoS	Denial of Service
ED	End Device
EDs	End Devices
EDA	Energy Depletion Attacks
FCntDown	Frame Counter Downlink
FCntUp	Frame Counter Uplink

FN	False Negatives
FP	False Positives
FE	Feature Extraction
FR	Feature Reduction
FS	Feature Selection
fNS	Forwarding Network Server
GW	Gateway
HBOS	Histogram-Based Outlier Score
HIDS	Host-Based Intrusion Detection System
IDS	Intrusion Detection System
IDPS	Intrusion Detection and Prevention System
IPS	Intrusion Prevention System
IoT	Internet of Things
JoinEUI	Join Server Identifier
JoinNonce	Nonce of Joining Server
JR	Join Request
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KLD	Kullback Leibler Divergence
LOF	Local Outlier Factor
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
LPWAN	Low Power Wide Area Network
LSNR	LoRa Signal-to-Noise Ratio
LSTM	Long Short Term Memory
MAC	Media Access Control
MITM	Man-In-The-Middle
ML	Machine Learning
MLlib	Machine Learning Library
NB-IoT	Narrowband IoT
NetID	Network Server’s Unique Identifier
NIDS	Network-Based Intrusion Detection System
NS	Network Server
NwkKey	Network Key
NwkSKey	Network Server Key
OCSVM	One-Class Support Vector Machine
PCA	Principal Component Analysis
RDD	Resilient Distributed Dataset

REST	Representational State Transfer
RF	Radio Frequency
RSSI	Received Signal Strength Indicator
SF	Spreading Factor
SIDS	Signature-Based Intrusion Detection System
SMO	Sequential Minimal Optimization
SNR	Signal-to-Noise Ratio
sNS	Serving Network Server
SVM	Support Vector Machine
SVD	Singular Value Decomposition
TCP	Transmission Control Protocol
TTN	The Things Network
vNS	Visiting Network Server
VM	Virtual Machine
WAL	Write-Ahead Logs
WebUI	Web User Interface
XGBoost	Extreme Gradient Boosting

Index

1 INTRODUCTION 1

1.1 APPROACH 2

1.2 OBJECTIVES 2

1.3 CONTRIBUTIONS 2

1.4 DOCUMENT STRUCTURE 3

2 STATE OF ART 5

2.1 INTRUSION DETECTION IN IOT NETWORKS 5

2.2 FUNDAMENTALS OF DATA SECURITY 6

2.3 LORAWAN IN IOT SYSTEMS 7

2.3.1 LoRaWAN Versions 7

2.3.1.1 LoRaWAN v1.0.4 vs v1.1 8

2.3.2 LoRaWAN Security Mechanisms 9

2.3.2.1 LoRaWAN Security Layers 9

2.3.2.2 Session Keys Derivation 10

2.3.2.3 End-Device Activation 10

2.3.2.3.1 Activation by Personalization (ABP) 10

2.3.2.3.2 Over-the-Air Activation (OTAA) 10

2.3.2.4 Replay Protection Mechanisms 11

2.3.2.5 Integrity and Authenticity Validation 12

2.3.3 LoRaWAN Networks: Security Vulnerabilities And Risks 13

2.3.3.1 Protocol Specification 13

2.3.3.2 Roaming 13

2.3.3.3 ABP Device Activation 14

2.3.4 LoRaWAN Networks: Attacks and Possible Detection Techniques 14

2.3.4.1 Jamming 14

2.3.4.2 Energy Depletion 15

2.3.4.3 Eavesdropping 15

2.3.4.4 Replay 15

2.3.4.5 Attack Setting 16

2.3.4.6 Wormhole 16

2.3.4.7 Sinkhole 16

2.3.4.8 Down-Link Routing 17

2.3.4.9 Destroy, Remove or Steal End-Device 17

2.3.4.10 Physical Tampering 17

2.3.4.11 Covert Channels 17

2.4 ALGORITHMS FOR INTRUSION DETECTION 18

2.4.1 Fundamentals of Machine Learning 18

2.4.1.1 Supervised Learning vs Unsupervised Learning 18

2.4.1.2 Deep Learning Fundamentals 18

2.4.2 Machine Learning Algorithms *19*
2.4.2.1 k-Nearest Neighbours 19
2.4.2.2 Local Outlier Factor 20
2.4.2.3 Logistic Regression 20
2.4.2.4 Decision Tree 20
2.4.2.5 Random Forest 21
2.4.2.6 Isolation Forest 21
2.4.2.7 Support Vector Machine 22
2.4.2.8 One-Class Support Vector Machine 22
2.4.2.9 K-Means 22
2.4.2.10 DBSCAN 23
2.4.2.11 Autoencoders 23
2.4.2.12 XGBoost 23
2.4.2.13 Histogram-Based Outlier Score 24
2.5 RELATED WORK 24
2.5.1 Machine Learning Based IDS Solutions *25*
2.5.1.1 IDS Solutions based on Deep Learning 26
2.5.2 Conclusions on Related Works *27*
3 ANOMALY-BASED INTRUSION DETECTION SYSTEM: IMPLEMENTATION 29
3.1 TOOLS FOR DATA PROCESSING 29
3.1.1 Fundamentals of Data Processing *30*
3.1.1.1 Stream Processing 30
3.1.1.2 Batch Processing 30
3.1.1.3 Hybrid Approach 30
3.1.1.4 Message Delivery Guarantee 31
3.1.2 Apache Kafka *31*
3.1.2.1 Kafka Streams 32
3.1.3 Apache Spark *32*
3.1.3.1 Spark Streaming 33
3.1.3.2 Structured Streaming 33
3.1.4 Apache Flink *33*
3.1.5 Apache Kafka vs Apache Spark vs Apache Flink *34*
3.2 EXPECTED ARCHITECTURE 36
3.3 DETAILED ARCHITECTURE 37
3.3.1 Input Dataset Structure and Characteristics *39*
3.3.2 Data Pre-Processing *41*
3.3.2.1 Feature Selection 41
3.3.2.2 Feature Extraction 43
3.3.2.3 Preparing Dataset to Create Model 44
3.3.2.4 Feature Assembling 45
3.3.2.5 Feature Scaling 46
3.3.2.6 Feature Reduction 46

3.3.2.6.1 Principal Component Analysis 47
3.3.2.6.2 Singular Value Decomposition 47
3.3.2.6.3 Autoencoders 48
3.3.2.6.4 Final Choice of Techniques 48
3.3.3 Data Processing 48
3.3.3.1 Model Training 48
3.3.3.2 Model Testing 49
3.3.4 Stream Processing Configuration Steps 51
3.4 USED TOOLS: FRAMEWORKS AND LIBRARIES 52
3.4.1 Frameworks 52
3.4.1.1 MLFlow 52
3.4.1.2 PySpark 52
3.4.2 Libraries 53
3.4.2.1 Scikit-Learn 53
3.4.2.2 NumPy 53
3.4.2.3 Pyod 53
4 EVALUATION AND RESULTS 55
4.1 RESULTS 55
4.1.1 Devices Used for Testing 56
4.1.2 Isolation Forest (SkLearn Implementation) 57
4.1.3 Isolation Forest (Custom Implementation) 58
4.1.4 One-Class SVM 61
4.1.5 Histogram-Based Outlier Score 62
4.1.6 Local Outlier Factor 65
4.2 ALGORITHMS COMPARISON 66
4.3 DISCUSSION 69
5 CONCLUSIONS AND FUTURE WORK 71
BIBLIOGRAPHIC REFERENCES 75
A LORA GATEWAY CONFIGURATION 83
B TRAINING PARAMETERS OF ML ALGORITHMS 87
B.1 ONE-CLASS SUPPORT VECTOR MACHINE 87
B.2 ISOLATION FOREST 88
B.3 HISTOGRAM-BASED OUTLIER SCORE 90
B.4 LOCAL OUTLIER FACTOR 91
B.5 DATASET STRUCTURE 91
B.5.1 RXPK Messages 92
B.5.2 TXPK Messages 96

Index of Figures

FIGURE 2.1 – ARCHITECTURE OF LORAWAN V1.0 9
FIGURE 2.2 – ARCHITECTURE OF LORAWAN V1.1 9
FIGURE 3.1 - PROJECT EXPECTED ARCHITECTURE 37
FIGURE 3.2 - DETAILED ARCHITECTURE: STREAM PROCESSING 38
FIGURE 3.3 - DETAILED ARCHITECTURE: MODEL CREATION 39
FIGURE 3.4 - STREAM PROCESSING SETUP 51
FIGURE 4.1 - RECALL (CLASS 1) FOR SKLEARN-BASED ISOLATION FOREST 57
FIGURE 4.2 - ACCURACY FOR SKLEARN-BASED ISOLATION FOREST 57
FIGURE 4.3 - F1-SCORE (CLASS 1) FOR SKLEARN-BASED ISOLATION FOREST 57
FIGURE 4.4 - TIME OF MODEL GENERATION FOR SKLEARN-BASED ISOLATION FOREST 58
FIGURE 4.5 - NUMBER OF PREDICTED POSITIVES AND NEGATIVES FOR SKLEARN-BASED ISOLATION FOREST 58
FIGURE 4.6 - RECALL (CLASS 1) FOR CUSTOM VERSION OF ISOLATION FOREST 59
FIGURE 4.7 - ACCURACY FOR CUSTOM VERSION OF ISOLATION FOREST 59
FIGURE 4.8 - F1-SCORE (CLASS 1) FOR CUSTOM VERSION OF ISOLATION FOREST 59
FIGURE 4.9 - TIME OF MODEL GENERATION FOR CUSTOM VERSION OF ISOLATION FOREST 60
FIGURE 4.10 - NUMBER OF PREDICTED POSITIVES AND NEGATIVES FOR CUSTOM VERSION OF ISOLATION FOREST 60
FIGURE 4.11 - RECALL (CLASS 1) FOR ONE-CLASS SVM 61
FIGURE 4.12 - ACCURACY FOR ONE-CLASS SVM 61
FIGURE 4.13 - F1-SCORE (CLASS 1) FOR ONE-CLASS SVM 61
FIGURE 4.14 - TIME OF MODEL GENERATION FOR ONE-CLASS SVM 62
FIGURE 4.15 - NUMBER OF PREDICTED POSITIVES AND NEGATIVES FOR ONE-CLASS SVM 62
FIGURE 4.16 - RECALL (CLASS 1) FOR HBOS 63
FIGURE 4.17 - ACCURACY FOR HBOS 63
FIGURE 4.18 - F1-SCORE (CLASS 1) FOR HBOS 63
FIGURE 4.19 - TIME OF MODEL GENERATION FOR HBOS 64
FIGURE 4.20 - NUMBER OF PREDICTED POSITIVES AND NEGATIVES FOR HBOS 64
FIGURE 4.21 - RECALL (CLASS 1) FOR LOF 65
FIGURE 4.22 - ACCURACY FOR LOF 65
FIGURE 4.23 - F1-SCORE (CLASS 1) FOR LOF 65
FIGURE 4.24 - TIME OF MODEL GENERATION FOR LOF 66
FIGURE 4.25 - NUMBER OF PREDICTED POSITIVES AND NEGATIVES FOR LOF 66
FIGURE 4.26 - RECALL (CLASS 1) FOR EACH ALGORITHM 67
FIGURE 4.27 - ACCURACY FOR EACH ALGORITHM 67
FIGURE 4.28 - F1-SCORE (CLASS 1) FOR EACH ALGORITHM 67
FIGURE 4.29 - TIME OF MODEL GENERATION FOR EACH ALGORITHM 68
FIGURE 4.30 - NUMBER OF PREDICTED POSITIVES AND NEGATIVES FOR EACH ALGORITHM 68
FIGURE A.1 - MIKROTIK LORA GATEWAY 83
FIGURE A.2 - SERVER CONFIGURATION FOR LORAWAN MESSAGE DESTINATION ("TESECMT") 84

FIGURE A.3 - DEVICE CONFIGURATION FOR LORA GATEWAY ("GATEWAY-1") 85

Index of Tables

TABLE 3.1 - COMPARISON BETWEEN APACHE KAFKA, APACHE FLINK AND APACHE SPARK.................................36
TABLE 3.2 - CONFUSION MATRIX STRUCTURE 49
TABLE 4.1 - NUMBER OF DEVICE SAMPLES AND TIME OF PRE-PROCESSING..56

1 Introduction

IoT is contributing to an exponential technological revolution that is improving the operational efficiency of organizations and companies and opening up new opportunities to adopt innovative technologies, having a great impact on the modern world as explained by Vicci (2024). By enabling millions of devices - such as smart house appliances like smart lamps - to continuously generate and exchange large amounts of data, IoT is revolutionizing multiple sectors such as manufacturing, smart cities, healthcare, sports and transportation, making people's daily routine more efficient through innovative technologies that rely on IoT systems to perform their functions, since these technologies use automation to improve decision-making and resource optimization.

There are several developed communication protocols to support IoT systems, and one of the most widely adopted protocols is LoRaWAN, which has been increasingly more adopted particularly to efficiently develop applications that require long-range low-power communications. However, previous studies have proved that IoT technologies also have several security vulnerabilities, especially LoRaWAN, due to its open, decentralized and distributed network architecture, as explained by Esteves et al. (2024). Even though these technologies bring a lot of benefits to the consumers, many times their security isn’t considered enough, which can easily make them a target for several cyberattacks with a great potential to compromise sensitive data, disrupt operations, or even take control of the IoT-connected devices. This can lead to potential severe damages such as data privacy violation, which compromise both individuals and organizations.

Because of this, it’s crucial to study what these security vulnerabilities are, what intrusions can take place in IoT systems, what techniques can be performed to detect these intrusions and finally how these techniques can be performed and applied. Therefore, it’s proposed to implement a solution which allows real-time detection of intrusions taking place in LoRaWAN-based IoT systems, so that organizations can get reported of any anomaly on time, before these intrusions potentially lead to several negative consequences such as service disruptions.

This document covers all the necessary steps to solve these problems, and what is the proposed solution to solve them, starting with this introduction chapter, that is organized in the following sections: section 1.1, which describes the proposed approach to solve the problem that was previously described in this chapter, section 1.2, which describes the objectives of this project, section 1.3, which describes previous projects that contributed for the development of the project and also the contributions that this project has provided, and section 1.4, which presents the main structure of this document.

1.1 Approach

In this project, the objective is to develop and evaluate an IDS, specifically for a LoRaWAN network. As explained by Fidalgo (2022), the developed IDS is a software that is constantly monitoring the LoRaWAN network to analyse every message coming from end-devices, to detect any intrusions coming from those messages, including novel threats. To develop this IDS, innovating ML algorithms will be applied. Innovative approaches, that move beyond traditional IDS solutions, will be employed to preserve data security, and dynamic adaptation mechanisms will be applied to effectively respond to changes in the network topology. The IDS evaluation will focus on its effectiveness in anomaly detection and mitigation of known and unknown attacks, based on evaluation metrics. This project aims to contribute to LoRaWAN networks' security and reliability, providing a robust foundation for intrusion detection in IoT infrastructures.

1.2 Objectives

The proposed solution in this project aims to accurately detect any anomalies coming from LoRaWAN messages that a LoRa gateway receives from many LoRaWAN EDs and forwards to a NS. The solution is an IDS installed near to the LoRa gateway, following an edge-computing based approach, which is applied to reduce latency associated with centralized servers, and enhance privacy and security in data transportation since data is not forwarded to a centralized server. After being trained with input data, the used ML algorithms must be able to accurately detect new LoRaWAN messages as normal or abnormal instances.

1.3 Contributions

This thesis was inspired by the solution proposed by Esteves (2023). The solution of this project followed an edge computing based approach adopted by that solution. Moreover, that solution provided the necessary background for the fundamentals of ML and the application of ML for LoRaWAN anomaly detection, and also a starting point to study the

most adequate ML algorithms to implement on the proposed solution in this project. That solution applied kNN, and that was a starting point for a deeper research on more adequate algorithms, and one of the tested algorithms in this solution, LOF, is inspired by kNN. The solution described by Esteves (2023) had previous support from Ferrovia 4.0, a project financed by the Innovation National Agency which provided all the necessary hardware tools for technology testing.

The implementation of this project is available in the GitHub repository published by Tavares (2025), which is open for improvements and updates as necessary in the future.

1.4 Document Structure

This document is organized in five chapters. The Chapter 1 is the Introduction, which corresponds to the current chapter. In Chapter 2, the state of art is described, where all the theoretical background that is necessary to implement this project is covered. In Chapter 3, all the details of the implementation of the project are described, including the expected architecture, the detailed architecture, the used tools, the necessary steps to perform the IDS described in detail, and the used algorithms. In Chapter 4, the results of the implementation of the developed IDS, the evaluation of those results and the project's limitations are described. Finally, the Chapter 5 covers all conclusions from this project, the finished tasks and challenges that were faced during the project development, and also future work that can be developed by other people.

2 State of Art

This chapter covers all the necessary theoretical background to design, implement and evaluate the projected IDS to implement. It starts by providing an overview of what are intrusions in IoT and how an IDS applies to IoT networks (section 2.1) and the most important fundamentals of data security (section 2.2) that must be considered to implement a robust and effective IDS. Moreover, the chapter explores LoRaWAN concepts that are relevant to develop a IDS specified to LoRaWAN network traffic anomaly detection, namely what is LoRaWAN (section 2.3), what are its existing versions (section 2.3.1), what security mechanisms it provides (section 2.3.2), what security vulnerabilities and risks exist on this protocol (section 2.3.3) and what attacks can take place on the protocol (section 2.3.4). The chapter also covers some ML algorithms that can be applied to the IDS to detect anomalies through network traffic analysis (section 2.4). Finally, the chapter covers some previous works (section 2.5) related to the current project, that serve as a background for the current project's implementation, as well as the gaps that exist on those works and how the current project solves them.

2.1 Intrusion Detection in IoT Networks

In IoT systems, as similarly explained by Zarpelão et al. (2017), intrusions refer to any unauthorized access or malicious activities conducted by non-authorized or non-authenticated users on interconnected devices, networks and systems. These intrusions can occur at different levels of a corresponding IoT architecture, typically composed of a wide range of sensors, at least one gateway, a network and an application. Intrusions can happen on a device level, on a gateway level, on a network level and on an application level. IoT systems can be targeted by several types of intrusions, such as DoS attacks, DDoS attacks, Packet Sniffing, Code Injection and Command Injection.

A typical IDS is composed of sensors, an analysis engine and a reporting system. Sensors gather data and send it to the analysis engine, which analyses the collected data and detects intrusions.

Moreover, Zarpelão et al. (2017) explain that an IDS can be classified with two categories: Network-Based IDS (NIDS) or Host-Based IDS (HIDS). NIDS monitors network traffic to detect intrusions, connecting to one or more network segments, and HIDS monitors traffic and malicious activities within the system of a computer device, not only monitoring network traffic like NIDS, but also system calls, running processes, file-system changes, inter-process communication and application logs.

Furthermore, Fidalgo (2022), Zarpelão et al. (2017) and García-Teodoro et al. (2008), explain that there are many categories of IDS based on their detection method. The first category is signature-based IDS (SIDS), which is the simplest approach. It's based on attacks' known signatures, and if the content of a packet is partially the same as the signature, an alert is triggered. The second category is anomaly-based IDS (AIDS), not only based on signatures but also on the behavioural analysis of the system's devices, learning the traffic behaviour and triggering an alert if the current behaviour is altered. There is still the specification-based IDS category, which represents approaches that detects deviations on network behaviour from specification definitions, where specifications are sets of rules and thresholds that define the expected behaviour for network components such as nodes, protocols and routing tables. Finally, there are the hybrid approaches, which combine the SIDS, AIDS and Specification-Based IDS approaches to maximize their advantages and minimize the impact of their disadvantages. The current project is equivalent to an AIDS, since it detects not only known attacks, but also unknown attacks, using ML models to learn network traffic patterns and detect any deviations from those patterns, classifying those as intrusions. The SIDS is the best approach to detect known attacks, but it's ineffective to detect new attacks and variants of known attacks, because a matching signature for these new attacks is still unknown. However, even though AIDS is the most effective strategy for attack detection in general, it usually leads to an high FP rate because, as also explained by Fidalgo (2022), Zarpelão et al. (2017) and García-Teodoro et al. (2008), it considers any deviation from the learned network traffic patterns as anomalies, even if these deviations are legitimate and not properly anomalies.

It's also explained by Fidalgo (2022) that IDS differs from IPS (or IDPS), because while IPS (and IDPS) detects and prevents intrusions, IDS only detects them, being passive on the system's communication, which means that it doesn't add latency on message analysis unlike IPS does.

2.2 Fundamentals of Data Security

To ensure data security in any information systems, including IoT systems, as explained by Qadir (2023), three characteristics from data used by these systems are fundamental: confidentiality, integrity and availability. Confidentiality ensures that data is protected

from unauthorized access or exposure, blocking access from unauthorized users who have no legitimate concern about it. Integrity ensures that data remains unaltered from source to destination, protecting data from potential tampering. Availability ensures that data remains reliable and easily accessible for use from authorized and authenticated users, focusing on the seamless continuity of information within the network.

2.3 LoRaWAN in IoT Systems

LoRaWAN is a specific protocol within the LPWAN family that uses LoRa modulation to provide long-range communication with low power consumption, as explained by LoRa Alliance (2017). LoRa is a long-range protocol that refers to the LoRaWAN physical layer (or MAC layer), and LPWAN is a protocol which refers to a more generic category of wireless communication technologies to enable long-range communication with low power consumption. As also explained by Adelantado et al. (2017), while this protocol is widely adopted by applications such as smart cities, video surveillance and smart transportation and logistics, it has also its limitations. Besides having an open and decentralized architecture which makes it vulnerable to security threats (as explained in Chapter 1), LoRaWAN has the effect of the duty cycle that significantly limits the ability of large-scale deployments on this protocol. Also, the TTN Fair Access Policy limits the time on air of each ED to a maximum of 30 seconds per day, which limits the protocol's flexibility to adapt to changes in the network topology and the environment, as well as to applications with tight latency or capacity requirements. A LoRaWAN gateway, which covers a range of tens of kilometres and able to serve up to thousands of EDs, must be carefully dimensioned to meet the requirements of each use case. To do so, it's crucial to properly define the combination of the number of EDs and select the adequate SF values and the number of channels.

2.3.1 LoRaWAN Versions

LoRaWAN protocol has many official versions since its release. The first version, v1.0, published by LoRa Alliance (2015), was raised in January 2015. The version v1.0.1, published by LoRa Alliance (2016b), was later released in February 2016. The version v1.0.2, published by LoRa Alliance (2016a), was released in July 2016 and most networks support it. The version v1.1, published by LoRa Alliance (2017), was released in October 2017, and it has made some security enhancements relatively to previous versions, being the most secure version until now. The version 1.0.3, published by LoRa Alliance (2018a), was later released in July 2018 and only added unicast and multicast to Class B devices and a new DeviceTimeRequest command. The version 1.0.4, published by LoRa Alliance (2020), is currently the latest official version of LoRaWAN,

released in October 2020, and provided security updates and improvements for Class B and Class C devices (section 2.3.2).

2.3.1.1 LoRaWAN v1.0.4 vs v1.1

LoRaWAN v1.0.4 and v1.1 are currently the most popular versions of LoRaWAN. However, LoRaWAN v1.1 uses a different architecture from the one used by v1.0.

LoRaWAN v1.0.4 has made several improvements over the previous v1.0.x versions. It made some clarifications on Class B and Class C operation modes as additive to Class A. Comparatively to LoRaWAN v1.1, LoRaWAN v1.0.4 is more simple to implement but it's less secure, since it does not discard FPorts above 224 and doesn't have the MAX_FCNT_GAP limit (explained in section 2.3.2.4) that LoRaWAN v1.1 has. Furthermore, LoRaWAN v1.0.4 implements only one global NS, while v1.1 has three types of NS that are part of a network operator: hNS, sNS and fNS. The hNS forwards messages between EDs and the AS, directly or via sNS or fNS, and sends MAC commands to the ED directly or via sNS or fNS; the fNS forwards messages exchanges between the ED and sNS or between EDs and hNS; and sNS forwards messages between hNS and EDs directly or via fNS and sends MAC commands to the EDs directly or via fNS. This separation of different NS from LoRaWAN v1.1 allows for a more secure and reliable key distribution to hNS and vNS in the communication between the Join Server (JS) and the ED. JS is a component added in LoRaWAN v1.1 that handles requests from the ED to join the network and generates every session security keys, improving the separation of trust between NS and AS. The better security from LoRaWAN v1.1 happens specially on Class A devices, according to the specification published by LoRa Alliance (2017) (section 2.3.2.5), since it has a better key management, using separate keys for encryption and data integrity check, and it also has separate frame counters (FCntUp and FCntDown) which makes replay attacks much more unlikely to happen (section 2.3.2.4). This security enhancement is especially important when dealing with large and complex networks that may need to scale across multiple network servers, and for deployments with strict security and reliability requirements, especially when supporting a broader range of Class B and Class C devices. In LoRaWAN v1.0.x, the ED communicates with its corresponding NS, and once the join procedure is completed, the ED will stablish the connection with the NS, since ED mobility isn't considered in v1.0.x. LoRaWAN v1.1 considers the mobility of the ED providing a new activation mode for it, which is called Roaming Mode (section 2.3.3.2). In most cases, v1.1 is the most secure version due to its best security. On the other hand, because v1.0.4 has less security requirements than v1.1 and introduces enhanced multicast support and better error handling, it can have a better network resilience and performance in some cases. To use v1.1, it's necessary to find and implement strategies to handle its increased complexity and give to the system a reasonable performance and

resilience. Figure 2.1 shows the architecture of LoRaWAN v1.0 and Figure 2.2 shows the architecture of LoRaWAN v1.1.

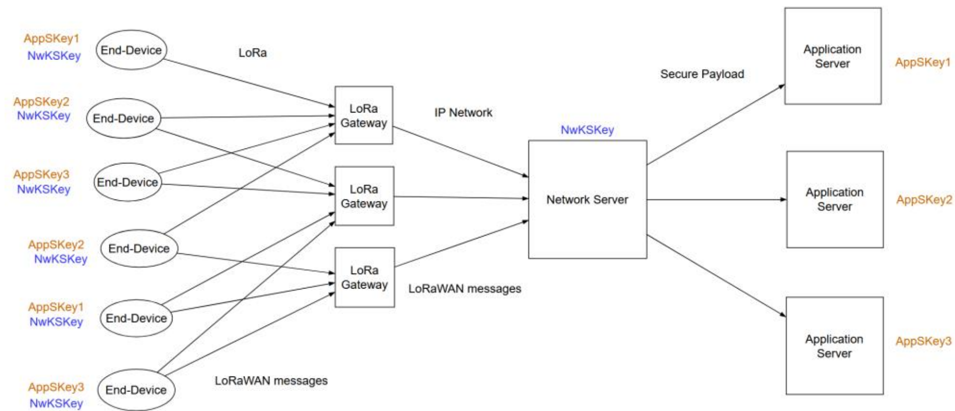

Figure 2.1 – Architecture of LoRaWAN v1.0

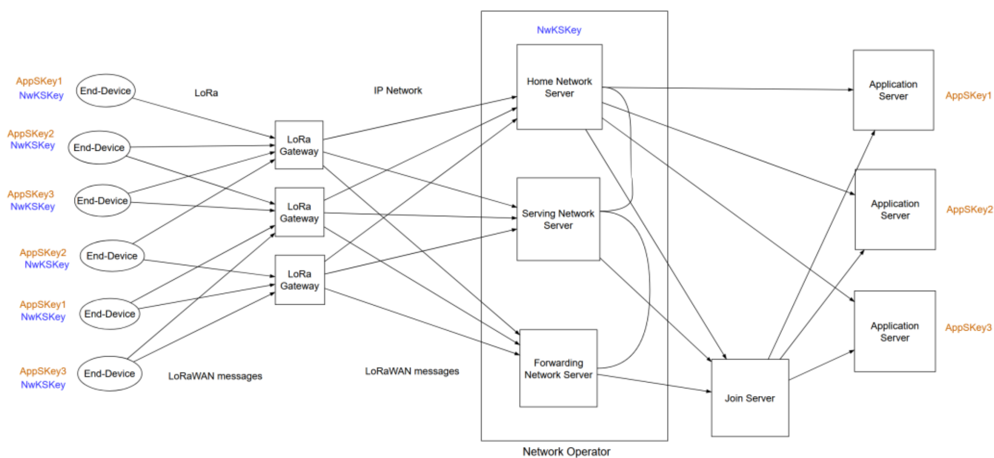

Figure 2.2 – Architecture of LoRaWAN v1.1

2.3.2 LoRaWAN Security Mechanisms

This subsection covers all security mechanisms that protect LoRaWAN from some known attacks, explaining what security layers LoRaWAN is composed of (section 2.3.2.1), how LoRaWAN session keys are derived (section 2.3.2.2), how LoRaWAN EDs can be activated (section 2.3.2.3), what security mechanisms are designed to protect LoRaWAN against replay attacks (section 2.3.2.4) and what security mechanisms validate the protocol's integrity and authentication (section 2.3.2.5).

2.3.2.1 LoRaWAN Security Layers

LoRaWAN has two security layers, as explained by Haxhibeqiri et al. (2018), Farrell (2018) and Semtech (2024):

- **Network Layer:** it's responsible for device identification based on Device Address (DevAddr), and also for integrity check;
- **Application Layer:** it encrypts and decrypts application payloads.

2.3.2.2 *Session Keys Derivation*

Session keys - NwkSKey and AppSKey - are used in LoRaWAN as illustrated on Figure 2.1, ensuring integrity (section 2.3.2.5) and confidentiality on the channel to secure the session. To derive these keys, LoRaWAN implements a cryptographic mechanism based on AES, as detailed by the specification published by LoRa Alliance (2017). This mechanism is based on a pre-shared key from which NwkSKey and AppSKey are derived.

2.3.2.3 *End-Device Activation*

Before performing the session key derivation process (section 2.3.2.2), EDs must be activated in order to get connectivity through a specific LoRaWAN network. Otherwise, the transmitted frames are silently discarded by the NS. In LoRaWAN v1.1, device activation is handled by Join Server (JS), that is combined with the NS. The JS manages security session keys (NwkSKey and AppSKey) and specifies the NS the user wants to work with. To ensure security, LoRaWAN implements two modes to activate an ED in the NS and AS, as detailed by LoRa Alliance (2017): Activation by Personalization (ABP), explained on section 2.3.2.3.1, and Over-the-Air Activation (OTAA), explained on section 2.3.2.3.2.

2.3.2.3.1 Activation by Personalization (ABP)

On this device activation mode, as detailed by LoRa Alliance (2017), before activation, the session keys (FNwkSIntKey, SNwkSIntKey, NwkSEncKey and AppSKey) and DevAddr are directly programmed in the device and stored in the server. When an ED is trying to communicate with the NS, it will send the messages directly. These messages are encrypted and signed, so that only the corresponding NS can read the messages. Session keys are used in every sessions until updated on the ED. AppSKey is stored in the AS and is common to all devices that use the same application and it's only known by the application. The other three session keys are NS keys, being stored in the NS. FNwkSIntKey and SNwkSIntKey ensure message integrity (section 2.3.2.5). NwkSEncKey is used to encrypt and decrypt the payloads with MAC commands to ensure message confidentiality. Despite being more simple than OTAA, ABP is less secure, because session keys are static. Vulnerabilities on ABP device activation mode are more detailed on section 2.3.3.3.

2.3.2.3.2 Over-the-Air Activation (OTAA)

Unlike ABP device activation mode, OTAA requires the ED to join the network through a Join Procedure to stablish connection, as detailed by LoRa Alliance (2017). The ED has to send a Join Request (JR) to the NS, which contains DevEUI, JoinEUI (only introduced in LoRaWAN v1.1) and DevNonce, and it's signed with AppKey, a 16-byte device-unique

key which is assigned by application owners to EDs. When the JR is received by NS, in case of LoRaWAN v1.1, the NS will forward the message to the JS, that will process the JR to check if the ED can be accepted or not. If the ED is not accepted, there will be no response. If the ED is accepted, the JS will generate AppSKey, session key derived from AppKey, and FNwkSIntKey, SNwkSIntKey and NwkSEncKey that are derived from NwkKey. These four keys are then sent from JS to NS. If the generation of these keys gets success, the NS generates a Join Accept (JA) message, which includes JoinNonce, NetID, DevAddr, DLSettings, RXDelay and CFList, and sends it to the ED. DevEUI is a global unique ED identifier, JoinEUI (only in LoRaWAN v1.1) is a global application identifier that uniquely identifies the JS that can assist in JR processing and session keys derivation, and DevNonce is a counter starting at 0 when the device is initially powered up and it's incremented with every JR, being used to prevent replay attacks (section 2.3.2.4). In LoRaWAN v1.1, MIC is calculated over all the fields in the JR message using the NwkKey, and the calculated MIC is then added to the JR message. JoinNonce is a device specific counter value (that never repeats itself) provided by the JS and used by the ED to derive the session keys FNkwSIntKey, SNwkSIntKey, NwkSEncKey and AppSKey, being incremented with every JA message, also to prevent replay attacks (section 2.3.2.4). Once again, MIC is calculated over all the fields, this time in JA message using JSIntKey (for LoRaWAN v1.1 devices). The calculated MIC is then added to the JA message. JA is then encrypted with NwkKey or JSEncKey using an AES decrypt operation in ECB mode, and the NS sends the encrypted JA message back to ED as a normal downlink. After this, JS sends AppSKey to the AS and the three network session keys to the NS, and ED decrypts the JA message using AES encrypt operation, and it uses AppKey, NwkKey and JoinNonce to generate session keys. In the end, DevAddr, FNwkSIntKey, SNwkSIntKey, NwkSEncKey, AppSKey are stored in the ED. With these mechanisms, attacks such as replay and eavesdropping are much more unlikely to happen in a OTAA-activated device than in a ABP-activated device, because session keys are derived through data encryption using AppKey and are unique to each ED. In LoRaWAN v1.0.x, because there is not a JS, the NS handles the JR itself.

2.3.2.4 Replay Protection Mechanisms

Frame counters (FCnt), as detailed by LoRa Alliance (2017), are responsible for protecting the NS from replay attacks. As the message counter is used in LoRaWAN to generate key streams, frame counters also contribute to confidentiality of the communication channel. In LoRaWAN v1.1, for each ED, there are two frame counters, FCntUp and FCntDown. FCntUp counts uplink messages in the ED and FCntDown counts downlink messages in the NS. In order to keep uplink and downlink messages in synchronization, there is a limit value called MAX_FCNT_GAP in LoRaWAN v1.1. If the difference between the numbers of uplink messages and downlink messages is greater

than MAX_FCNT_GAP, subsequent frames are discarded. If the counter overflows, it will be started from 0 again, after resetting. However, it's necessary to guarantee that FCnt overflow is avoided, because if that happens, FCnt can be repeated within the same session and session keys can be reused between one captured real session and one fake session, opening possibilities for a replay attack. Also, in LoRaWAN v1.1, an ED keeps track of the last observed JoinNonce value, specified as JoinNonce_last, and if a JA is replayed, it will be discarded by ED based on the JoinNonce_last value. ED only accepts JoinNonce values which are incremented compared to JoinNonce_last value. In the same way, NS keeps track of the last DevNonce used by each ED, specified as DevNonce_last, and ED only accepts JoinNonce values incremented compared to DevNonce_last value, discarding the remaining values. The timestamp in the LoRaWAN message header and the generation of new session keys during the activation procedure on OTAA explained on section 2.3.2.3.2 are also mechanisms used by LoRaWAN to prevent replay attacks.

2.3.2.5 Integrity and Authenticity Validation

As detailed by LoRa Alliance (2017), LoRaWAN uses a cryptographic message integrity code (MIC) to sign all messages exchanges between ED and NS and provide integrity check on the MAC header and payload data. This protects LoRaWAN messages from unauthorized tampering caused by attacks such as MITM, Bit-Flipping, and other attacks such as Beacon Synchronization, ACK spoofing and False Join Packets. MIC is calculated using the NwkSKey and AES-CMAC method, a block cipher-based message authentication code algorithm whose output is appended at the end of the LoRaWAN message. When uplink messages arrive at the NS, the server will first check the message integrity, and if it passes the check, it sends the message to the AS. Unlike LoRaWAN v1.0.4, that only has one type of NwkSKey, LoRaWAN v1.1 has two types of NwkSKey: Serving Network Session Integrity Key (SNwkSIntKey) and Forwarding Network Session Integrity Key (FNwkSIntKey). On LoRaWAN v1.1, SNwkSIntKey is used to verify MIC of downlink messages and FNwkSIntKey is used to calculate MIC or half of the MIC of all uplink data messages to guarantee data integrity. Uplink messages MIC rely on these two session integrity keys to allow forwarding NS in a roaming setup to be able to verify at least half of the MIC field. Similarly, AppSKey is used for encrypting and decrypting the payload of application data. To mitigate ACK spoofing on LoRaWAN v1.1, a new MIC calculation can take place, involving part of the FCnt (section 2.3.2.4) of the acknowledged frame. But it's necessary to avoid compromised integrity of communication channel between NS and AS to minimize the risk of unauthorized tampering.

2.3.3 LoRaWAN Networks: Security Vulnerabilities And Risks

Despite all LoRaWAN security mechanisms explained on section 2.3.2, there are still some gaps on LoRaWAN security that can be exploited by attackers to perform some possible attacks, and those vulnerabilities are covered on this subsection, namely vulnerabilities on LoRaWAN specification (section 2.3.3.1), on Roaming mode provided by LoRaWAN v1.1 (section 2.3.3.2) and on ABP device activation mode (section 2.3.3.3).

2.3.3.1 Protocol Specification

In LoRaWAN, if the keys are compromised or stolen, the LoRaWAN network becomes completely vulnerable and exposed, because all LoRaWAN security mechanisms come from them, as detailed by LoRa Alliance (2017). Noura et al. (2020) stated that there is the risk of the keys being stolen during the message transmission, during key management phase, during session keys generation and when these keys are stored on ED. When LoRaWAN keys are compromised, it's very hard to change the AES keys across all end-devices. Furthermore, accessing device keys increases the attacker's chances to compromise the other keys. Also, the encrypted message and the secret key have the same length, and the GW sends periodically a beacon to the NS, which can allow the attacker to employ these beacons to, for example, send beacons at a higher rate than the original one. Furthermore, in LoRaWAN, there is no handshake mechanisms between a GW and an ED, which means that an uplink message sent by an ED is normally received by every GW, and every GW can transmit downlinks to every ED. In case of downlink messages, in order to not overload the network, only one GW is selected to transmit the downlink message, and for communication quality optimization, the NS selects as the downlink GW the one that received the last uplink with the best radio parameters. This guarantees that the chosen GW provides the most favourable conditions for communication. However, this opens possibilities for an Attack Setting, explained on section 2.3.4.5.

2.3.3.2 Roaming

LoRaWAN v1.1 has introduced a new activation mode to end-devices called Roaming, as detailed by LoRa Alliance (2017) and Donmez & Nigussie (2018), which allows end-devices to send data through other networks by stablishing connectivity beyond the range of their home network. By performing this mode, the ED, already activated, can switch from one operator to another and maintain their connectivity with a LoRaWAN network even being outside of the cover area of their home network. Despite all these advantages and innovations, Roaming also presents its disadvantages, and one of its biggest disadvantages is its security vulnerabilities. To activate the roaming mode, the vNS needs to access to the ED's session information, which the roaming ED sends in a

plain text message, and this makes the system more vulnerable to attacks such as sniffing, replay and eavesdropping, when the attacker captures the exchanged information.

2.3.3.3 ABP Device Activation

Yang et al. (2018) and Aras et al. (2017) approached with more details vulnerabilities on this device activation mode. As explained on these studies and also on section 2.3.2.3.1, each ED has unique session keys that are static and stored in the server, and when the ED tries to communicate with the server, it sends the messages directly. Therefore, the ED being reset, it will reuse the FCnt value from 0 with the same keys. This opens more possibilities for a eavesdropping (section 2.3.4.3) or replay attack (section 2.3.4.4). Attacking an ABP-activated ED take less time as reset and counter overflow work if the attacker has the ability to reset the ED. To minimize these vulnerabilities in LoRaWAN v1.1, ABP-activated ED must use non-volatile memory to store the frame counters, and an ABP-activated ED's frame counters must never be reinitialized. Since no rekeying is possible in ABP, if the mechanism of non-volatile storage is correctly implemented, the possibilities for a replay or eavesdropping attack are minimized, since the attacker has to wait for a counter overflow. To mitigate these attacks on ABP activation mode, changing the session keys regularly is an helpful strategy. However, it is still recommended to minimize the use of ABP activation mode when possible.

2.3.4 LoRaWAN Networks: Attacks and Possible Detection Techniques

Considering the security vulnerabilities detailed on section 2.3.3, there are still attacks that can take place on LoRaWAN networks, as studied by Noura et al. (2020) and Zarpelão et al. (2017), but they can also be detected by the IDS through various strategies. The following subsections detail attacks that are possible on LoRaWAN, namely how they work and how they can be detected.

2.3.4.1 Jamming

Jamming Attack, as studied by Ruotsalainen et al. (2022) and Qadir (2023), is a DoS attack that interferes in the LoRaWAN physical layer, affecting the availability of the ED by targeting communication between the ED and the NS. It consists of sending radio signals in the same frequency of an ongoing radio transmission of an ED, which mask legitimate signalling, leading to services' disrupt or even bringing down a network. To detect this attack, Danish et al. (2018) approached a LoRaWAN-based IDS that used KLD and Hamming Distance to compute the difference between the behaviour of the random numbers sent from end-devices to the NS as part of the JR packet during a normal LoRaWAN operation and the behaviour of these random numbers when a jamming attack is performed.

2.3.4.2 Energy Depletion

Energy Depletion Attack (EDA), as studied by Proto et al. (2024), is a device-level attack that aims to deplete the ED's battery until this becomes unavailable, contributing to reduce the battery lifetime of the ED and having the potential to make an entire sensors' network unavailable. It is predominantly characterized as an attempt to force the ED to increase its transmission or retransmission, generate downlink packets and overflow the reception stage. This attack usually derives from other attacks such as flooding or jamming, since these attacks also force the ED to consume more energy than expected. EDA can still be a silent or ghost attack, which means that, on this case, it uses up the sensors' energy without generating network traffic, usually compromising the ED by exploiting a firmware or API vulnerability, and this category of EDA is much harder to detect through an IDS. Several studies implemented different techniques to detect attacks like energy depletion that directly affect ED's energy consumption. Other studies have implemented an intrusion detection architecture that analyses the energy consumption of sensors to detect EDA in LoRaWAN. However, the energy consumption analysis for EDA detection is not an easy challenge because this depends of the variation of parameters like the number of EDs, transmission frequency and noise frequency.

2.3.4.3 Eavesdropping

Eavesdropping affects the confidentiality of the LoRaWAN network. On this attack, the attacker passively monitors the network's traffic in order to gather information about data transmissions. This attack is only possible on ABP-activated end-devices, as explained on section 2.3.3.3. However, it's relatively challenging to detect this attack since it's a passive attack, which means that it does not affect the network traffic. OTAA provides security mechanisms to make eavesdropping much more unlikely to happen on OTAA-activated devices, as explained on section 2.3.2.3.2.

2.3.4.4 Replay

On the replay attack, the attacker repeats available data transmitted by malicious entities, resending captured messages from the ED. The attacker uses communication frequencies and channels to sniff data from transmission between ED and GW. As explained on section 2.3.3.3, on ABP device activation mode, when the FCnt overflows, the same FCnt values are reused with the same static session keys. This vulnerability allows the attacker to intercept messages from a previous session with larger counter values and reuse them in the current session. As a result, the attacker disrupts the communication between ED and NS, needing only a traffic sniffer and a LoRa transmitter to replay messages. OTAA device activation mode provides security mechanisms that make replay attacks much more unlikely to happen (section 2.3.2.4), especially when the attacker tries the selective RF jamming method, as explained by Honfoga et al.

(2024). To detect replay attacks on ABP, the IDS should analyse the network traffic patterns and compare the current traffic with these patterns.

2.3.4.5 *Attack Setting*

Spadaccino (2023) explains that Attack Setting consists of replaying an uplink message in close proximity to a geographically distant GW, and it can manipulate the NS's deduplication phase, causing it to select the remote GW for sending the downlink response. As a result, the device targeted by the attack would be unable to receive expected downlink messages from the NS, including critical ones like the JA after a JR. To detect attacks from this nature, an anomaly detector performing past downlink paths analysis could be used to seek clusters of late packets or unusual GW activity to detect ongoing attacks.

2.3.4.6 *Wormhole*

Stanco et al. (2024), Honfoga et al. (2024), Ruotsalainen et al. (2022), Torres et al. (2021) and Aras et al. (2017) explain that wormhole is an attack that affects the device availability. It consists in packet sniffing and replaying the sniffed packets. A malicious device, called sniffer, captures packets from a legitimate ED and transmits them through signals to another device, called jammer, to notify that it captured the packet, in order to prevent the packet from reaching the destination. In this way, the packet remains active for validation. The captured message can be replayed at any time. The sniffer and the jammer must be physically distant from each other so that the sniffer can receive the intended packets. Besides that, for this attack to succeed, the attacker has to find a way to block legitimate packets, and a reactive jammer needs to be placed in the network, preferably close to a gateway. When this attack is combined with others, it becomes an even more serious threat. An example of consequence of this attack is that critical alarm messages can be jammed and never reach the intended target, preventing authorities from being notified from existing anomalies. Because this attack is normally a combination of various attacks, to detect it, the same techniques that are used to detect replay or jamming attacks can be applied.

2.3.4.7 *Sinkhole*

Qadir (2023) explains that Sinkhole Attack also affects the availability of the device, especially when combined with other attacks. It consists of directing the network traffic to a specific node, where the attacker will promote a specific route in the network and infect the network nodes to use that route. This attack can be detected and mitigated using an IDPS, therefore the IDS implemented on this work is adequate to detect this attack, comparing the current network traffic with the patterns learnt by the implemented models.

2.3.4.8 Down-Link Routing

When a GW is not authenticated, it can be targeted by a Down-Link Routing attack, as explained by Qadir (2023). On this attack, the ED sends an uplink message to an authenticated GW, and at the same time the attacker eavesdrops on the transmission channel between the ED and the authenticated GW and replays the eavesdropped traffic to a different network through the compromised GW that is not authenticated. The NS will validate the replayed packet and update the downlink routing path for the GW. The result is that the NS will receive duplicate packets from both gateways. Therefore, the methods to detect this attack might be the same methods to detect replay attacks.

2.3.4.9 Destroy, Remove or Steal End-Device

In this attack, as explained by LoRa Alliance (2017), Qadir (2023) and Butun et al. (2018), since LoRaWAN root keys are uniquely generated for each ED during fabrication or before deployment, revealing a single root key (key from which session keys are generated) will only compromise the data stored in the specific device, and not any other information transferred all over the network. Hence, this attack will compromise the availability, confidentiality and integrity of the stored data. Thus, to detect this attack in LoRaWAN, it's necessary to analyse aspects such as what is coming from the ED in the LoRaWAN message and if it mismatches the patterns learnt by the ML algorithms, or if the ED was sending messages and suddenly stopped sending them.

2.3.4.10 Physical Tampering

LoRa Alliance (2017), Stanco et al. (2024) and Butun et al. (2018) explain that all LoRaWAN session keys must be securely stored. Otherwise, an attacker can extract and reuse these keys to impersonate an ED, compromising the hardware, disrupting communications and accessing to sensitive data for manipulation. As previously explained on section 2.3.3.1, all LoRaWAN security mechanisms rely on the keys. By extracting the keys, the attacker can modify the device firmware, replace its components or install spyware to gain control over the ED. Besides storing the keys in secure components, firewalls and access control are also valid strategies to minimize the chances for this attack to happen. However, all these security mechanisms don't completely eliminate the possibilities of this attack to happen. To detect this attack through an IDS, the ED traffic patterns must be analysed to see if the current traffic mismatches them, or if the ED was sending messages and suddenly stopped sending them.

2.3.4.11 Covert Channels

Ruotsalainen et al. (2022) and Hou et al. (2022) explain that Covert Channels is an attack which is only applicable to LoRaWAN v1.0.x. This attack targets the LoRaWAN physical

layer, using a transmission medium to transmit sensitive information such as secret keys. The goal of building a covert channel is to get information out without affecting the performance of a regular LoRaWAN channel and avoid being detected by LoRaWAN security mechanisms. Therefore, detect this attack using an IDS might be challenging.

2.4 Algorithms for Intrusion Detection

This section provides the theoretical background of ML applied on this project. It starts by explaining the fundamentals of ML that are important to understand the mechanisms inside the used algorithms for this project (section 2.4.1). Then, it explains some algorithms used on previously implemented IDS that can be adequate for this project's implementation (section 2.4.2).

2.4.1 Fundamentals of Machine Learning

ML is a subset of AI that allows machines to learn the best case scenario based on a given dataset of stored information, applying a adequate optimization strategy in a data-driven way, as explained by Cuomo et al. (2020). ML algorithms use statistical methods to identify patterns and anomalies in large datasets, enabling security analysts to detect previously unknown threats, as explained by Ozkan-Okay et al. (2024).

2.4.1.1 Supervised Learning vs Unsupervised Learning

Obaido et al. (2024) explained that, unlike unsupervised learning, that deals with unlabelled data and the corresponding model learns and finds patterns in data without knowing the outputs, in supervised learning, data are labelled and used to train the model so that the produced results are the most optimized possible with the desired outputs defined as labels.

2.4.1.2 Deep Learning Fundamentals

Deep Learning (DL) is a subset of ML based on deep neural networks, artificial neural networks with a structure comprised within input and output layers. DL models can comprise non-linear relations, which are more complex than linear relations and generate computational models where the object is expressed in terms of the layered composition of primitives. A deep learning model has always one input layer, that receives the input data used by the model, at least one hidden layer and one output layer which shows the final output of the model's predictions. Each layer, except the input layer, receives the output produced by the previous layer as its input, uses it to calculate its predictions and produce the output that will be used as the next layer's input.

Neural Networks generally use an activation function to calculate the outputs based on the inputs. Rasamoelina et al. (2020) explains that the activation function determines the

type of separation border in the inputs' space, and it can generate smoother and more continuous separation boundaries. It applies a non-linear transformation on a layer's input data, on a linearly separable problem, allowing the network to learn and execute more complex and non-linear tasks. Each neuron in the neural network applies its activation function to the input's weighted sum before transmitting the output to one of the next layers' neurons. A problem is linearly separable when there are weights and biases that define a linear boundary between the region of the excitatory response (the region in which the neuron is activated) and the region of the inhibitory response (the region in which the neuron is inhibited). In other words, linear separability makes it possible to correctly separate different types or classes of data by means of a linear boundary, which can be, for example, a straight line or a hyperplane.

2.4.2 Machine Learning Algorithms

This section details some algorithms that might be adequate to implement on this project's ML model. It explains how they function and how they could be applied on the used LoRaWAN dataset to detect anomalies.

2.4.2.1 k-Nearest Neighbours

kNN (k-Nearest Neighbours) is a supervised learning algorithm used for both classification and regression problems, as explained by Esteves et al. (2024) and Esteves (2023). This algorithm calculates the average similarity value of the k nearest neighbours (with highest similarity scores) and sets a threshold, and only when the average similarity value is above the threshold, is the packet considered normal, as explained by Liao & Vemuri (2002). It indicates the likelihood that a data point will become a member of one group or another based on which groups the data points nearest to it belong to, as explained by Mari et al. (2023). In other words, it indicates how close a packet is to rely inside the standards of a considered normal packet. kNN is a popular choice for network intrusion detection due to its simplicity and effectiveness. Its performance also depends on the chosen value of k. For the current project, kNN would be used for classification problems, using previously known data to determine the proximity of new data comparing to previous data. In this solution, this algorithm would get parameters from the LoRa gateway, namely its location, parameters of the received message, such as RSSI and SNR, and the LoRaWAN message itself, and it would learn the network traffic patterns and compare the current network traffic with the learnt network traffic patterns. Then, if the current traffic deviates from the traffic patterns, an alarm would be triggered to indicate an anomaly. Esteves et al. (2024) explained that implementations like kNN can adapt dynamically to new attack signatures. kNN can also be used to impute missing values from dataset features, in data pre-processing stage.

However, this algorithm performs intensive computations since it's based on distance metrics, which makes it not adequate for very large datasets.

2.4.2.2 Local Outlier Factor

LOF is an unsupervised ML algorithm inspired by kNN (section 2.4.2.1), as it considers the local density deviation of a point from its k nearest neighbours, assigning higher scores to instances with lower local densities. Alghushairy et al. (2020) and Breunig et al. (2000) describe that LOF calculates the degree of isolation of each data point of the training dataset, allowing the model to detect local outliers even in complex datasets. However, this algorithm generally requires a long execution time for large-scale or real-time data streams and, similarly to kNN, LOF is sensitive to the manually defined value of the 'k' parameter. But, despite this disadvantage, LOF is still much faster and efficient than other algorithms described in this state of art and remains one of the most effective techniques to detect subtle anomalies in high-dimensional data.

2.4.2.3 Logistic Regression

Logistic Regression is a supervised learning algorithm. This algorithm provides a binomial response variable which helps to estimate the probability of an event taking place. It uses logistic functions to help to measure that probability, and it's based on a dataset of independent variables. These characteristics make this algorithm very useful for binary classification problems. In fact, this algorithm can be particularly adequate for the current IDS. On this solution, it would determine the probability of a packet to be anomalous, as similarly proposed by the authors in Esteves et al. (2024), despite not proposing this specific algorithm to do so:

- $X = 0\%$: the message is classified as normal;
- $0\% < X <= 50\%$: the message requires a human review to check if it indicates an anomaly or not;
- $X > 50\%$: the message indicates an anomaly and is classified as abnormal. A flag in the database is added.

2.4.2.4 Decision Tree

Decision Tree is a supervised algorithm that works incrementally building learning models in form of a tree, splitting up the dataset based on appropriate conditions on its features, as explained by Cuomo et al. (2020). It's easy to interpret and it can handle multi-output problems that create a tree-like model of decisions and their possible consequences. It analyses the training data to identify the most important features for classification. It then creates a tree structure, where each internal node represents a feature, and each leaf node represents a class label. Data is recursively split at each internal node, based on the feature's values until it reaches the leaf node corresponding

to the predicted class label. Decision Tree is useful to classify packets based on their parameters' values and their values' learnt patterns and it's able to model non-linear relations, but it tends to be prone to overfitting, and Random Forest solves this problem, as explained on section 2.4.2.5.

2.4.2.5 Random Forest

Random Forest is a supervised algorithm that can be used to handle both classification and regression problems. On this algorithm, N data subsets derived from the original dataset are randomly generated, and these subsets function as decision trees. Then, each decision tree calculates its predictions, and after all decision trees calculate their predictions, all results are aggregated. The model's final prediction is the predicted target voted by the highest number of decision trees. Abdelaziz et al. (2024) explain that this algorithm is very versatile, accepting a big variety of input data, including numeric and categorical features and features with missing values, and the algorithm also stands out by its robustness since it's less prone to overfitting than algorithms such as Decision Tree explained on section 2.4.2.4. Overfitting occurs when the model fits too well to a specific dataset but shows a poor performance when handling new data. Random Forest solves this problem by calculating the average of the predictions of several randomly generated decision trees, which lessens the influence of trees that could cause overfitting. Jaiswal & Samikannu (2017) approach techniques to use Random Forest to handle both classification and regression problems, and they also consider Feature Subset Selection techniques for data pre-processing, so that the model predictions are more efficient and accurate, which is significantly important in a large dataset. In fact, on the current project, Random Forest would be used for classification problems, classifying packets as anomalous or non-anomalous.

2.4.2.6 Isolation Forest

Isolation Forest has many similarities with Random Forest, but these two algorithms differ in how they create decision splits, as explained by Cao et al. (2024). While Random Forest selects a feature and a threshold that maximizes the information gain or minimizes the Gini impurity, which depends on the class labels of the data points, Isolation Forest randomly selects a feature, followed by the random determination of a split value within the range of a chosen feature, without considering class labels. Isolation Forest is based on the idea of isolating instances in a dataset. Each generated tree is independent and is formed in the same way. The tree-building process creates elements splits until all elements are isolated. During training, a forest of isolation trees is created, where each tree randomly splits the data until each element is isolated, with the depth of the tree being limited. The isolation depth of a point determines its probability of being an anomaly, since anomalous points are isolated more quickly. In the evaluation stage, the

depth of the leaf node where a point ends when examined in each tree is calculated, and an anomaly score is assigned to the point based on the average of these depths. The higher a point's anomaly score, the higher the probability of that point to be anomalous. Hence, this algorithm is perfectly adequate for anomaly detection in this solution. However, this algorithm is not natively supported by PySpark, which means that it's necessary to integrate other tools with Spark to use this algorithm on this solution.

2.4.2.7 Support Vector Machine

SVM is also an option for binary classification, to classify a LoRaWAN message as anomalous or non-anomalous. As explained by Nayak et al. (2015), this algorithm separates various classes of the training dataset with a surface that maximizes the margin between the classes. In other words, SVM maximizes the model's capability of generalization. This is the objective of the Structural Risk Minimization principle, which allows the minimization of a border on a model's generalization error, rather than minimizing the quadratic average error in the training dataset. However, this algorithm might not be adequate for classification in large scale datasets due to its complexity and its excessive computational cost, and also because the training process is seen as a quadratic programming problem to find a separation hyperplane which implies a density matrix n x n, where n is the number of points in the dataset. Therefore, creating a SVM model requires a lot of computational time and memory space for large datasets.

2.4.2.8 One-Class Support Vector Machine

One-Class SVM is an unsupervised algorithm used for outlier detection that works by estimating the support of a high-dimensional distribution. Ghiasi et al. (2024) explain that this algorithm is an extension of the SVM algorithm (section 2.4.2.7) to unlabelled data, and it basically projects the input data into an high-dimensional space through a kernel function and builds a hyperplane for classification. They also explain that this transformation allows the projected data to be relatively linear, making it easy to distinguish between normal and anomalous data. Point classification would be performed with the help of a decision function, g(x), which determines whether a new point is within the hyperplane side of inliers (normal messages which are labelled by the algorithm as +1) or outliers (anomalous messages which are labelled by the algorithm as -1). The training involves minimizing a loss function that balances the maximization of the margin around the normal instances and the minimization of instances outside this margin.

2.4.2.9 K-Means

Suyal & Sharma (2024) explain that K-Means is an unsupervised learning algorithm, that divides examples of dataset into K clusters, where K is a previously chosen number.

That division is based on a centroid-based partitioning. The process of clustering allows that each cluster has examples with similar properties, and each cluster has examples that are different from the examples of the other clusters. Cuomo et al. (2020) and Valtorta et al. (2019) proposed solutions which use k-means to cluster LoRaWAN EDs according to their radio and network behaviour, where the EDs in a same cluster have similar activity profiles in radio resources usage.

2.4.2.10 DBSCAN

Kaliyaperumal et al. (2024) explains that, similarly to k-means, DBSCAN is a clustering algorithm that categorizes data into groups considering their characteristics. However, DBSCAN is a non-parametric and density-based algorithm that stands out by clustering closely packed data points of varying shapes without requiring a predetermined cluster count like k-means does. DBSCAN would be used to partition LoRaWAN dataset examples into high-density clusters and merge those sharing common neighbouring points, ensuring flexibility and balance in the training dataset. It generates sub-datasets within each cluster, each with its own classifier. During testing, labels are assigned based on the most relevant cluster's classifier.

2.4.2.11 Autoencoders

Berahmand et al. (2024) explain that AE belongs to the category of unsupervised DL algorithms that consist of neural networks trained to reproduce its input to its output. An AE has an hidden layer that defines the code used to represent the input. An AE neural network has two parts: the encoder function and the decoder function, which attempts to reproduce the input with minimal reconstruction error. While AE effectively learns useful data features for representation learning and dimensionality reduction, it can't perfectly replicate inputs and only approximate patterns seen in training data. This makes it useful for feature extraction but computationally intensive. Additionally, if the training dataset doesn't represent the testing data well, this algorithm might complicate learning rather than improve generalization.

2.4.2.12 XGBoost

Chimphlee & Chimphlee (2024) explain that XGBoost is a supervised learning technique that belongs to the family of gradient-boosted decision trees ML algorithms, and is used for both classification and regression problems. This algorithm minimizes the difference between the predicted values and the actual values using a loss function that calculates the divergence between them. On training, it improves predictions iteratively by adding decision trees that reduce the error from previous iterations while using regularization to prevent overfitting. XGBoost is known for its speed, accuracy, ability to handle missing data and scalability though parallel processing, making it adequate for large datasets like

the dataset used on this project. Its boosting method focuses on misclassifying data points, enhancing predictive performance. However, it can be prone to overfitting if hyperparameters aren't properly defined, difficult to interpret with many trees, and may require longer training times with careful parameter optimization.

2.4.2.13 Histogram-Based Outlier Score

HBOS is an unsupervised algorithm that finds outliers in datasets without prior training, scoring records in linear time. Goldstein & Dengel (2012) explain that this algorithm finds the outliers by estimating densities using histograms, where two different methods can be used for numerical features: static bin-width histograms and dynamic bin-width histograms. The first is the standard method to build histograms, using k bins with the same width over the value range. In this case, the frequency of samples of each bin is used as a density estimator (height of the bins). The dynamic bin-width is determined by sorting the values and group a fixed number of N / k successive values into a single bin where N is the number of total instances and k the number of bins. Bins covering a larger interval of the value range have less height and represent a lower density, since the width of a bin corresponds to the first and the last value and the area is the same for all bins. An often-used rule of thumb is to set the number of bins (k) to the square root of the number of instances (N).

HBOS stands out by its fast computation in large datasets, which is important in this solution, because it's constantly receiving messages in real-time, and the models are periodically re-trained, which means that the data used for training will become increasingly larger. However, the results highlighted by Goldstein & Dengel (2012) show that, despite being effective at detecting global outliers, HBOS fails to detect local anomalies. However, this behaviour is expected, because histograms can't model local outliers with their density estimation, and HBOS treats each feature independently, assuming that the values distribution across the entire feature space is homogeneous. As a result, it fails to consider interactions between features and can't model dense clusters surrounded by slightly less dense neighbourhoods, which is often where local anomalies appear. It's the opposite of what happens in local outlier detection methods like LOF (section 2.4.2.2), which compare the density of a point with the density of its neighbours, making them more adequate to identify local deviations.

2.5 Related Work

In modern network infrastructures, implementing IDS solutions is fundamental to identify and mitigate anomalies in real time, because those anomalies can disrupt operations and compromise the reputation of organizations. This section reviews relevant literature on IDS for IoT environments, with a particular focus on applying ML techniques (section

2.5.1), and it finishes with the conclusions of the research, detailing research gaps (section 2.5.2).

2.5.1 Machine Learning Based IDS Solutions

IDS Suricata was proposed by Fidalgo (2022). It consists on an IDS installed on each GW on a LoRaWAN network, implemented with Lua and Python programming languages to access data inserted and stored in the relational, scalable and SQL-based database CrateDB, using kNN (section 2.4.2.1) to analyse network traffic patterns due to its simplicity, efficiency and the nature of the used variables. This article was an inspiration for the solutions described by Esteves (2023) and Esteves et al. (2024), which added the edge computing approach to their solutions and integrated it with Suricata, comparing its performance with the performance of the centralized server approach, being one of the big inspirations for the current project. This IDS is implemented and evaluated in datasets in a LoRaWAN messages that were previously captured from a LoRa gateway, and provides a pre-processing module whose parameters are extracted by Python scripts. However, solutions proposed by Fidalgo (2022) and Esteves (2023) lack the implementation of other ML algorithms, and the current project studies more adequate algorithms for anomaly detection. Danish et al. (2018) proposed a LoRaWAN-based IDS (LIDS) that uses KLD and Hamming Distance to detect jamming attacks launched during ED's Join Procedure, using the approach described on section 2.3.4.1. KLD achieved a detection rate of 98% and Hamming Distance achieved a detection rate of 88%, with 5% of false positives, but while this solution only approaches jamming attack detection, the current project considers the detection of any traffic anomalies in LoRaWAN networks that might correspond to any attack. Cuomo et al. (2020) proposed an IDS mainly focused on k-means, LSTM and Decision Trees. They proved that k-means can be used to cluster EDs according to their behaviour, as explained on section 2.4.2.9. The window approach is also used, to predict when EDs send packets, analysing the time between an ED's consecutive transmissions through the sliding window technique. Decision Tree is used to predict the packet's next arrival time, as explained on section 2.4.2.4. LSTM networks are used to predict when a ED sends a new packet However, LSTM requires more computational resources than Decision Tree and, during the testing of the proposed solution, Decision Tree had a better performance than LSTM. This article only approaches three ML algorithms, and the proposed solution is not specified to LoRaWAN networks, unlike the current project that approaches more algorithms for anomaly detection. Proto et al. (2024), proposed a lightweight and energy-efficient IDS developed to detect EDA (section 2.3.4.2), a category of attacks that force LoRaWAN EDs to consume much more energy than expected by increasing its transmission or retransmission, generating downlink packets and overflowing the reception stage. The solution applies distance metrics to detect anomaly behaviours in the energy

consumption patterns of LoRaWAN EDs. However, it only approaches EDA as possible attacks in LoRaWAN. Chandra & Singh (2024) proposed an IDS also based on ML algorithms but it's more generic, unlike the current project that is more specific to LoRaWAN. For a better efficiency while executing ML models, data pre-processing was applied using PCA and FS, using HyperOpt library for hyperparameters optimization, written in Python. After pre-processing, data was classified using Decision Tree, Extra Tree, Random Forest and XGBoost. Chimphlee & Chimphlee (2024) provided a deeper study of XGBoost for network intrusion detection, but it doesn't have a practical implementation. Samantaray et al. (2024) proposed a NIDS to detect anomalies in IoT networks, using more ML algorithms such as Decision Tree, Random Forest and kNN, but the implementation isn't specified to LoRaWAN and doesn't provide so much ML algorithms like the current project does. The proposed solution in Rashid et al. (2024) uses k-means and DBSCAN for clustering and SMO for classification, but it's also not specified to LoRaWAN. Sarker et al. (2020) proposed IntruDTree, a ML-based intrusion detection system that first considers the ranking of security features according to their importance and then builds a tree-based generalized intrusion detection model based on the selected important features. Besides being an effective model, it also minimizes the model's computational complexity by reducing the feature dimensions, which is an important strategy to improve the model's results and avoid the "curse of dimensionality".

2.5.1.1 IDS Solutions based on Deep Learning

There are several previous works that present intrusion detection models that implement Deep Learning algorithms to detect anomalies on IoT networks. Vigneswaran et al. (2018) proposed a DNN-based IDS to compare the performance of DNNs with the traditional ML algorithms. DNN model was trained using KDDCup-'99' dataset, and different neural network depths were tested (1 to 5 hidden layers). ReLU activation function was used to avoid the greedy problem. The 3-hidden layer model performed the best, and the DNN architecture had the best performance (greater precision, recall and F1-score) on attack detection, comparing to the other used ML algorithms. Hore et al. (2024), proposed Deep ResNIDS, which classifies malicious packets using a Deep Learning model which identifies patterns of known attacks, and it uses autoencoders to identify zero-day attacks. To improve the detection of new attacks, the framework includes one-shot transfer learning, allowing the model to quickly adapt to new network threats without compromising the detection of known attacks. One-shot learning contributed for improvements on detecting zero-day, out-of-distribution and adversarial attacks but the current project's solution uses more robust algorithms specifically for LoRaWAN network intrusion detection. However, the current project implements ML algorithms rather than mathematical models and it's specified to LoRaWAN and not generalised to IoT. Despite the disadvantages of the studies, all of them prove that Deep

Learning has a great potential to make the difference on a robust NIDS for anomaly detection.

2.5.2 Conclusions on Related Works

There are a lot of previous network-based IDS solutions applied on IoT to detect anomalies, so it wasn't possible to cover all of them on this document. However, most of these implementations are generalised to IoT networks, and less than 10 of the IDS solutions referenced on this document are specified to LoRaWAN, and no deep-learning based solution is specified to LoRaWAN. Additionally, no implementations of IDS detecting anomalies covering all possible attacks in LoRaWAN networks were found on a first research. The currently available LoRaWAN-based IDS solutions only use a few ML algorithms, or they only cover one or two LoRaWAN attacks. In fact, previous proposed solutions do not cover a wide range of attacks and IoT technologies, as also explained by Zarpelão et al. (2017). Therefore, the current project has a big contribution for LoRaWAN security, not only to study more ML algorithms, but also to implement a more robust and efficient IDS solution. This work provides the first systematic comparison of ML algorithms for LoRaWAN anomaly detection using real gateway data, establishing empirical baselines for future research. However, despite being generalised to IoT, the found IDS solutions are a useful background for the current project.

3 Anomaly-Based Intrusion Detection System: Implementation

After carrying out the state of art, the focus shifted into implementing the proposed solution, and the main goal was to develop a solution capable of receiving LoRaWAN messages in real-time and detect anomalies, following an edge-computing based approach. This chapter covers every detail of the implementation of the IDS. The chapter starts by covering some tools that could be used for large-scale data processing (section 3.1), as well as some fundamentals of data processing (section 3.1.1). Then, section 3.2 covers the expected architecture of the project, delivering a more generic insight of the project structure and its main components. Then, section 3.3 covers the project's detailed architecture, which describes the details of the implementation of the component IDS from the expected architecture. Finally, section 3.4 details all used frameworks and libraries to develop the IDS according to the defined requirements.

3.1 Tools for Data Processing

For this project, it was necessary to use a framework capable of processing the input data used on the implemented solution, as well as providing a good balance between performance, scalability, fault tolerance, architecture and ease of use, as studied by Gimaletdinova (2024). Performance is evaluated by factors such as execution speed and computational efficiency. Scalability is the ability to handle large volumes of data and support an increasing number of nodes in a distributed cluster. Fault tolerance mechanisms must be supported in order to allow the system to keep available even when facing faults of computational clusters, considering that these clusters are constantly running and their components might fail. In order to find and choose a tool that meets these expectations, three tools were studied: Apache Kafka (section 3.1.2), Apache Spark (section 3.1.3) and Apache Flink (section 3.1.4). In order to comprehend these

frameworks' mechanisms, it was necessary to have a deep understanding of the fundamentals of data processing, explained on section 3.1.1.

3.1.1 Fundamentals of Data Processing

This subsection explains some fundamentals that are necessary to understand in order to choose the most adequate tool to process data and implement the most effective and efficient solution, namely stream processing (section 3.1.1.1), batch processing (section 3.1.1.2), hybrid processing (section 3.1.1.3) and message delivery guarantee semantics (section 3.1.1.4).

3.1.1.1 Stream Processing

Kleppman (2017) explains that event streaming consists on capturing data in real time from different event sources, and storing that data as event streams permanently for later retrieval, and also manipulating existing event streams and routing events to different destination technologies as needed, in order to guarantee a continuous data flow. Stream processing involves unbounded data streams, which means that input may never end, and it is necessary to continuously process data as it arrives.

3.1.1.2 Batch Processing

Kleppman (2017) explains that, in batch processing, bounded data streams are processed. Data is grouped into small batches, and each group of data is processed by independent batch jobs. Each batch job processes a part of data in parallel and independently from the other workers. The average processing time is decreased as the batch grows, but the first batch members will be subject to a higher latency as it has to wait for the batch to gather sufficient data. With batch processing, it's possible to handle a lot of previous data and analyse it deeply and perform operations such as sorting data, computing global statistics or producing a final report summing the complete input.

3.1.1.3 Hybrid Approach

The idea of this project is to follow an hybrid approach, exploring the benefits of both stream and batch processing, because there is the need to provide an high accuracy of data analysis through batch processing using the provided to train the models, but, at the same time, a fast data processing through stream processing to process and classify new data points in real time. The used datasets on this project are very large but it's also important to get the processing results with as low latency as possible, and neither batch processing nor stream processing can fully meet both goals on their own. Also, in real case scenarios, data is constantly being sent and forwarded to the solution, which emphasises the importance of stream processing, but, at the same time, it's very important to implement mechanisms to process uplink messages and downlink

messages independently and in parallel, because these two types of messages have different structures and will be processed in different ways, which emphasises the importance of batch processing. As explained by Hasani et al. (2014), Lambda Architecture is a useful background that was proposed by Nathan Marz in 2013 and it's an effective way to combine batch processing with stream processing. Data is split in two parts, one for batch processing, which handles large sets of data at once, and another for stream processing, which handles data as it comes in.

3.1.1.4 Message Delivery Guarantee

Vitorino (2022), when proposing an architecture called IoTMapper for a platform that gathers metrics about heterogeneous LPWAN networks, explained that there are three possible semantics used to guarantee message delivery:

- **At-most-once processing:** a message can be processed at most one time, which means that, without implementing fault tolerance mechanisms, messages can be lost when a system fault takes place;
- **At-least-once processing:** a message can be processed at least one time, which means that, without implementing fault tolerance mechanisms, messages can be duplicated when a system fault takes place;
- **Exactly-once processing:** a message is always processed exactly one time. This is the ideal scenario for fault tolerance, because messages are neither lost nor repeated.

3.1.2 Apache Kafka

Apache Kafka, as explained by Apache Kafka (2024), is a distributed event streaming platform used for real-time reliable exchange of large quantities of data. It's based on the Publish-Subscriber software pattern, on which producers are client applications that publish events to Kafka, sending data, and consumers subscribe to those events using data stored on those events. Kafka broker is the core module of Kafka, being responsible for data storage and processing. It is a unique instance of Kafka and is part of a bigger cluster. Kafka is executed as a cluster of one or more servers, and clients allow the development of distributed applications and microservices that read, write and process streams of events in parallel, at scale and with fault tolerance. Data streams in Kafka are unbounded and sorted lists called topics, where publishers publish their data and consumers retrieve that data after subscribing to the corresponding topic. Topics are split in smaller parts called partitions, which gives Kafka more scalability, because many machines can process each one of the partitions, and the possibility of parallel consuming by different consumers, where each consumer consumes data from a different partition. Each topic keeps its own indexing system called offset, which means that there is no information about registry order on partitions. Kafka Connect is a

component that receives and sends data flow, simplifying communication between systems. Kafka provides native support for Apache Spark and Apache Flink, and data availability, redundancy and fault tolerance through data streaming replication. For metadata management, Kafka supports Apache Raft (KRaft), a consensus protocol that eliminates Kafka dependency from Apache Zookeeper, simplifying Kafka's architecture. Kafka also provides a ML library (kafka-ml) which supports TensorFlow and PyTorch, two of the most popular Deep Learning existing frameworks.

3.1.2.1 Kafka Streams

Kafka Streams, as explained by Apache Kafka (2024), is a client library responsible for developing applications and services where input and output data are stored in Kafka clusters, proving a strong integration with Kafka technology through simplified development and deployment of Java and Scala applications on the client side. It's a simple and lightweight library that can execute, in parallel, multiple instances of a Kafka application, instead of requiring a cluster for separate processing. It supports the exactly-once semantics, previously explained on section 3.1.1.4. Kafka Streams has strong dependencies with Kafka, but only has dependencies with Kafka, as an internal messaging layer. It provides the stream processing required primitives through two APIs: Streams DSL, a high-level API with a simpler syntax for faster developing, and Processor API, a low-level API with a more complex syntax for a higher control of details.

3.1.3 Apache Spark

Apache Spark, as explained by Apache Spark (2024), is a popular framework used for large-scale data processing that supports both stream processing and batch processing, but it's mainly focused on batch processing. Spark is based on the MapReduce programming model provided by Apache Hadoop, doing all its job in the form of table transformations. Spark provides high-level APIs in Java, Scala, Python (PySpark), R and SQL (Spark-SQL), and uses in-memory computation for task executions, querying data on disks only when memory is full, allowing a faster processing than on other tools such as Apache Hadoop, that writes data on disk at each operation. Stream processing is performed using micro-batching, what allows the developer to specify the item count or the time interval for each micro-batch, even though these parameters can also be set by default. Spark does not have its own file management system, so it needs an external system to manage its files, such as Apache Hadoop. Spark follows a master-worker architecture where a driver application interacts with a cluster manager which manages several worker nodes in which executors run. The executor can run on the same machine also called a horizontal cluster or on separate servers which is an example of vertical clustering. Spark includes two streaming APIs: Spark Streaming (section 3.1.3.1) and Structured Streaming (section 3.1.3.2).

3.1.3.1 Spark Streaming

Spark Streaming, as explained by Apache Spark (2024), is a legacy Spark streaming engine that receives input data streams and divides them into batches, that are then processed by the Spark engine to generate the final stream of results in batches. It is an extension to the main Spark API that enables scalable, high-throughput and fault-tolerant stream processing of live data streams. Spark Streaming provides a high-level abstraction called DStream, a continuous stream of data represented by a continuous series of RDD that, as explained by Zaharia et al. (2012), are an immutable and distributed set of objects partitioned in a Spark data cluster that supports in-memory storage and can be cached in iterative algorithms, allowing stream processing organization as a sequence of stateless and deterministic batch computations across short time intervals, and contributing for Spark's improvement of speed, throughput and resilience.

3.1.3.2 Structured Streaming

Structured Streaming, as explained by Spark (2024), is an updated version of Spark API built on Spark SQL engine that also provides scalability and fault-tolerance. However, it's more user-friendly than Spark Streaming, being more adequate for streaming applications and pipelines. This system implements the exactly-once semantics through checkpointing and WAL, providing a fast, scalable and fault-tolerant stream processing without the user having to reason about streaming. It reads the latest available data from the streaming data source, processes it incrementally to update the result when there is new data, and then discards the source data.

3.1.4 Apache Flink

Apache Flink, as explained by Flink (2024), is an open-source framework for distributed data processing for stateful and continuous computations for data unbounded and bounded streams. It provides native support for both batch processing and stream processing, but it's mainly focused on stream processing. Flink is executed in parallel in a distributed cluster, and instances from an operator are executed independently in separate threads, which contributes for load balancing. Flink stands out by its high capacity of processing data in real time, allowing applications to respond quickly to data changes. It follows a master-worker architecture built by a JobManager, a ResourceManager, a Dispatcher and TaskManagers. The JobManager has responsibilities such as task scheduling, failure recovery and checkpoint coordination. The ResourceManager controls resource supply and allocation in the Flink cluster, managing task slots, which are the unit of resource scheduling in a Flink cluster. The dispatcher offers a REST interface to submit Flink applications and launches a new

JobMaster for each submitted job, running the Flink WebUI where information about job executions can be visualized. A JobMaster is responsible for managing the execution of a single JobGraph, and each JobMaster oversees the execution of a particular JobGraph, which stands for a Flink task. Multiple jobs can run simultaneously in a Flink cluster, each having its own JobMaster.

Flink supports two types of stream processing: Timely Stream Processing, which considers the event occurrence order rather than their processing order, and considers the ability to reprocess historical data with the same used code to process data in real time, and Stateful Stream Processing, where the way how a event is processed depends on the event of the previously processed events. Flink also supports two types of data structures: DataStream, the main data structure on the Flink ecosystem which is an immutable data collection on a Flink program, and DataSet, which are only applicable in bounded datasets, where data are already available during execution, and share the same attributes as DataStreams. Flink also provides FlinkCEP, a module for complex event processing that can be used to detect event in a infinite event stream, and allows the specification of customized patterns, which describe, in a abstract way, the detected data and also triggered actions upon matching event sequences.

3.1.5 Apache Kafka vs Apache Spark vs Apache Flink

As previously explained and studied by Cavallin & Orpana (2024), Könemann (2022), Chaves et al. (2023), Aouria & Mezati (2024), Adesina & Adesina (2024) and Ibtissame et al. (2024), these three tools provide native support for ML libraries, which is one of the requirements to implement the current solution.

Apache Kafka is the best tool for data distribution, offering the lowest latency on data processing, initialization and recovery, as proved by studies approached by the document described by Vitorino (2022). On the other side, it does not provide native support for batch processing, which means that it would be necessary to integrate Kafka with other tools like Flink or Spark to support this mechanism. However, the idea is to follow an hybrid approach as explained on section 3.1.1.3, so Apache Kafka would only be adequate for this project if it was integrated with tools like Flink and Spark that support batch processing, but this option would provide a greater complexity and be more time-consuming.

The choice between Apache Spark and Apache Flink took into consideration specific requirements for this hybrid approach and these tools' support, performance, scalability and fault tolerance. As previously explained, both Apache Flink and Apache Spark provide native support for batch and stream processing, as also explained by Gimaletdinova (2024), but while Flink is mainly focused on stream processing, treating a batch as an finite stream, Spark is mainly focused on batch processing, treating streaming applications as a special case of micro-batches. Because of these aspects,

Flink is generally more adequate for systems that require immediate insights, and Spark is generally more adequate for systems when efficient processing of large amounts of data, a good data understanding and a great data analysis is a priority. When it comes to fault tolerance, Spark Streaming uses RDD as previously explained on section 3.1.3.1, which allows the recomputing of lost partitions in case of failures, and supports lineage, a directed acyclic graph of transformations applied to RDD, which facilitates the recovery of lost data by re-executing transformations on available partitions. On the other hand, Flink takes advantage of the exactly-once semantics approach (section 3.1.1.4) to use a distributed snapshot technique to capture the state of the processing pipeline at regular intervals, and when a failure takes place, Flink rolls back to the latest consistent snapshot, resuming processing from that point. It replicates the input data across multiple nodes, ensuring that the failure does not lead to data loss. Hence, Flink and Spark both provide valid and consistent approaches to handle system faults and data loss.

Gimaletdinova (2024) compared Spark, Hadoop MapReduce and Flink on big data parallel processing, and proved that Apache Flink has an higher processing speed than Apache Spark. García-Gil et al. (2017) stated that, while Spark is manually optimized, allowing the user to control partitioning and memory caching and using the JVM's heap memory to manage all its memory, Flink implements a thoroughly iterative processing in its engine based on cyclic data flows, where a join operation can be planned as a complete shuffling of two sets, or as a broadcast of the smallest one. Moreover, Gimaletdinova (2024) stated that Apache Spark provides more operations and tools than Apache Flink and it has a greater maturity, project size, market share and community. Even though Spark has greater latency on data processing, batch processing is typically used for handling data in large datasets like the one used on this project, and it's indispensable in scenarios where comprehensive data analysis is required. The IDS implemented on this project required a deeply understanding about what messages anomalies were detected, what kind of anomaly was detected in those messages, in what messages no anomaly was detected and have a deep understanding about the evaluation metrics. In fact, Flink has a large growth margin when it comes to maturity, stability and frameworks' support, and its novel concepts have influenced advancements and improvements on Spark's latest versions. Also, since the objective is to implement ML algorithms on this project, it was taken into consideration that SparkML provides more ML algorithms and libraries than FlinkML and Kafka-ml, and that Python was chosen to develop the ML model and only SparkML supports Python. Considering all these aspects, Apache Spark was chosen for implementing this project. Table 3.1 provides a detailed resume of the comparison of these three tools based on the most relevant comparative criteria.

Table 3.1 - Comparison between Apache Kafka, Apache Flink and Apache Spark

	Apache Kafka	Apache Flink	Apache Spark
Data Processing Mechanisms	– Real-Time Data Distribution as focus – Stream Processing	– Stream Processing as focus – Batch Processing	– Stream Processing – Batch Processing as focus
Processing Speed	– Much faster than Flink and Spark on data processing, initialization and recovery	– Faster than Spark, due to its architecture focused on stream processing, but slower than Kafka	– Slower than Flink and Kafka, due to its architecture focused on batch processing
Optimization	– Needs to be manually optimized	– Provides an integrated optimizer	– Needs to be manually optimized, but Spark documentation provides support for that
Development Complexity	– Generally, not as complex as Flink, since it provides a low-level API to process complex events – Requires Kafka Streams, Flink or Spark for Complex Event Processing	– More complex to develop than Spark since it provides a less robust documentation and a less mature and developed API	– Simpler to develop than Flink and Kafka since it provides a more robust documentation and a more mature and developed API
Documentation Support	– Provides more learning resources than Flink	– Doesn't provide as much learning resources as Spark and Kafka	– Provides more learning resources than Flink
ML Libraries and Algorithms Support	– Kafka-ml provides TensorFlow and PyTorch for Deep Learning only	– FlinkML doesn't provide as much libraries and algorithms as Spark MLLib, which would limit the effectiveness and robustness of the IDS	– Spark MLLib provides more ML libraries and algorithms than Kafka-ml and FlinkML, contributing for the robustness and effectiveness of the IDS
API Programming Languages (ML)	– Kafka-ml is used on Python	– FlinkML can be used in Python or Java	– APIs available in Python, R, Java, Scala and SQL
Maturity	– More mature on message ordering, and integrate well with Flink and Spark	– Flink API not as mature as Spark API – Java API is more mature than Python API	– Spark is more mature than Flink, providing high-level APIs (such as RDD) to optimize query execution and improve performance
Computational Efficiency	– Needs Flink / Spark to optimize use of resources on stream processing	– Uses less computational resources than Spark	– Uses more computational resources than Flink due to its in-memory based approach
Fault Tolerance Mechanisms	– Data streaming replication	– It uses a distributed snapshot technique	– It uses RDD

3.2 Expected Architecture

The project's expected architecture is illustrated on Figure 3.1. End-Devices send messages to the NS using LoRa gateways as intermediaries, NS receives those messages and also has connection with an AS. Esteves (2023) explains that the NS is responsible for validating the messages, verifying the MIC, attributing network addresses and make the relation between DevEUI and the network address available for the other

components. The AS is responsible for processing and interpreting data coming from the NS, which is often visualized on the AS. On the other hand, IDS receives those messages directly from the LoRa gateway, processes them, retrieves necessary data from a local data source to help on message processing, classifies the received LoRaWAN messages as anomalous or normal messages, and stores the results on the used local data source to be possible to visualize the results later.

This architecture proposes an edge computing approach, where the IDS is installed near the LoRaWAN gateway. Edge computing is a decentralized data processing approach which strategically deploys computing resources close to data sources. This approach has advantages when comparing to the centralized server approach, such as lower latency, enhanced privacy and security, and a better real-time analysis and throughput, allowing facilitating detection and response to potential intrusions. However, it also presents challenges, since edge devices may have limited computing resources compared to centralized servers. To address these limitations, it’s necessary a careful optimization of ML models to implement a solution which operates efficiently within these limitations. Esteves et al. (2024) explain that, on this approach, LoRa gateways need to have packet decryption capabilities, but the latency associated with the NS is reduced. However, if the adequate pre-processing techniques are not applied, ML algorithms might become computationally intensive, adding too much latency, perhaps more than the centralized server approach. Hence, it’s necessary to perform adequate pre-processing techniques to reduce latency associated with ML algorithms processing, and these techniques will be explained later.

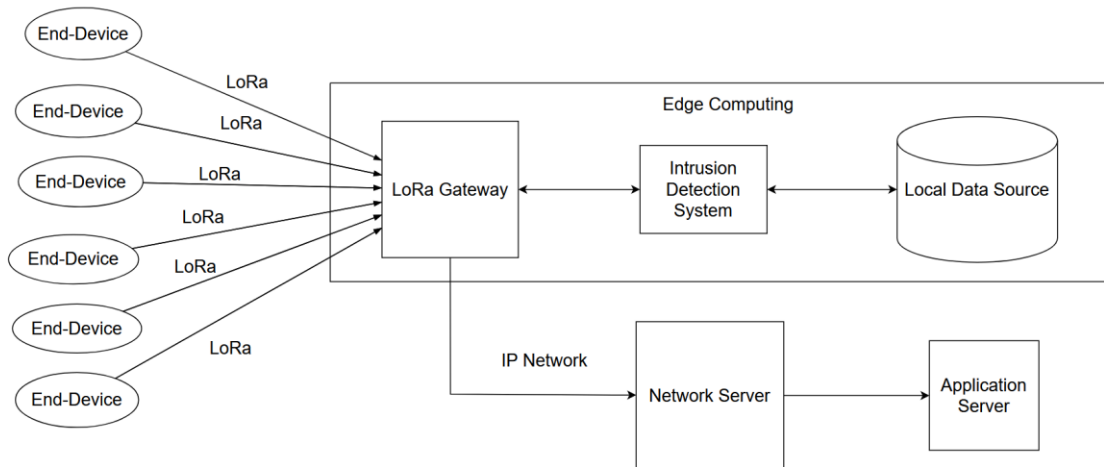

Figure 3.1 - Project Expected Architecture

3.3 Detailed Architecture

The detailed architecture is very important on any project, since it describes the implementation details. On this project, it specifies each step performed to apply ML

using an input dataset and use the created ML models for anomaly detection. Then, it details each one of the steps performed to implement the detailed architecture, breaking down into each technique studied and used in pre-processing, and what ML algorithms were used and tested.

Figure 3.2 and Figure 3.3 describe the detailed architecture in a generic context. Figure 3.2 describes the real-case scenario, where real LoRaWAN messages are coming from a LoRa gateway, in real time, to be processed by Spark. Figure 3.3 explains how the IDS creates models from the provided static datasets taking advantage of Spark batch operations, without the logic of real time message processing. Each one of the steps to implement these architectures will be explained in more details below in the document.

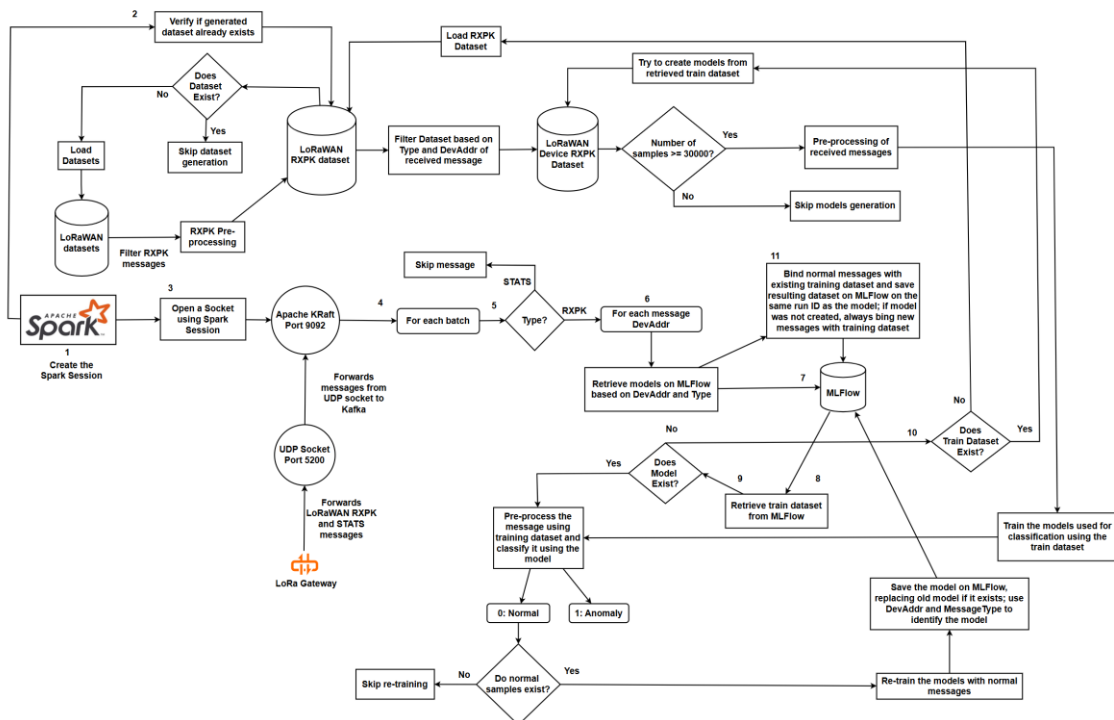

Figure 3.2 - Detailed Architecture: Stream Processing

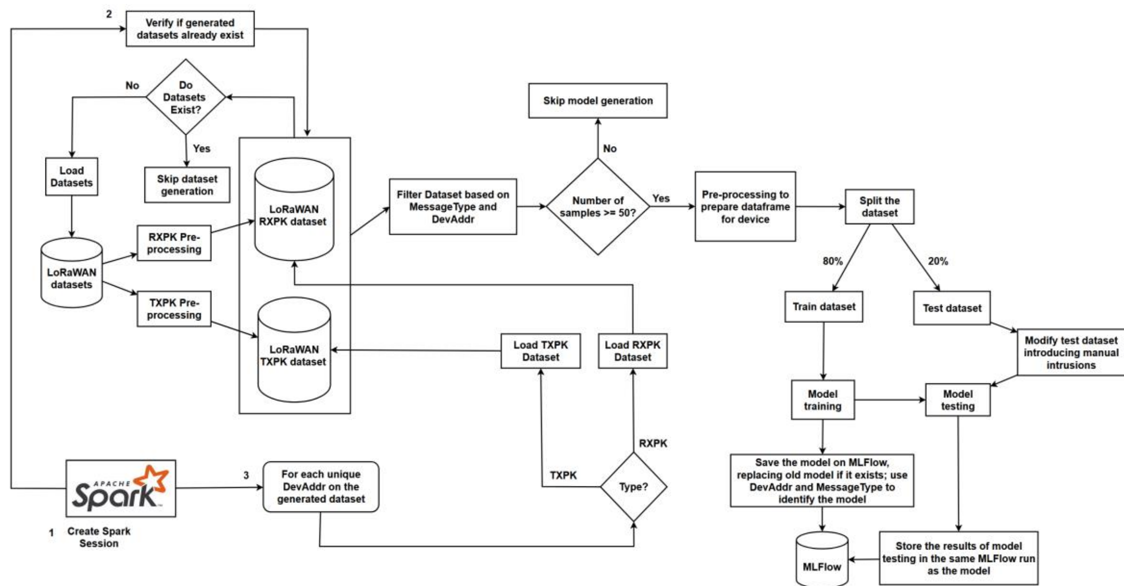

Figure 3.3 - Detailed Architecture: Model Creation

The idea is to use PySpark (section 3.4.1.2) to receive LoRaWAN messages in real time and train the model where the input training data is mostly composed by normal messages. After the model being trained, it must be able to detect anomalies in the test dataset that correspond to instances which are different from most training data, as explained by Goldstein & Dengel (2012). Hence, an unsupervised approach will be used in training, where only a single dataset without labels is given and an appropriate ML algorithm is used to train the model, so that the model, during testing, is able to identify outliers based on their feature values only. This is the main objective of the IDS: classifying messages in real time. To do so, as previously explained in Chapter 2, PySpark takes advantage of stream processing operations (Structured Streaming) to receive LoRaWAN messages in real time from a configures LoRa gateway, but also from the batch processing operations, to process static datasets with previous messages that will be used to train the models for each device, that will be used to classify the messages in real time.

3.3.1 Input Dataset Structure and Characteristics

The provided static datasets are a set of .log files that contain every LoRaWAN messages captured from a LoRa gateway in June 2020, in JSON format. The gateway was operated by the Lisbon Polytechnic Engineering School (ISEL), in Portugal, and the messages captured by the gateway came from thousands of EDs. In total, 1,610,557 RXPK messages and 15,438 TXPK messages are inside all of these files, showing an imbalance between RXPK and TXPK messages. Note that, from those 1,610,557 RXPK messages included on that dataset, many do not come from EDs that have joined the network. Only 608,519 of those messages come from devices were actually joined, as

only these contain a DevAddr. Consequently, only messages that contain a DevAddr were used to train and test the ML models, since the objective was to build models dedicated to learn the network traffic patterns of a specific device and a specific type of messages (RXPK of TXPK) coming from that device. This means that, for a specific device, a maximum of 2 models can be created. Separating the traffic in this way allows each model to learn the unique behavioural patterns of a single device and message type. As a result, the model can detect anomalies that are specific to that device, instead of being influenced by traffic from multiple devices, where values that are anomalous for one device might appear normal for another devices, which results on potentially losing important anomalies from that device. Additionally, RXPK and TXPK messages are intentionally handled by different models because they contain different types of information and have distinct traffic patterns. Keeping these message types isolated ensures that each model learns the appropriate patterns for its category and can identify traffic deviations more efficiently.

From the relevant parameters that are inside the datasets, Esteves et al. (2024) explained some types of anomalies that the models are expected to detect considering the nature of the datasets. They stated that RSSI and LSNR offer quantitative measures of signal quality and strength, enabling the detection of anomalies in transmission patterns. SF, a fundamental parameter in LoRa modulation, enables identification of changes in the ED configuration. The length of the payload (PHYPayload) can also indicate anomalies, since deviation from expected message sizes may indicate unauthorized transmissions. By creating models able to detect these types of traffic deviations, the IDS is expected to be able to maintain the integrity of LoRaWAN payload encryption. When it comes to parameter correlations, Esteves et al. (2024) showed that "chan" (channel) and "freq" (frequency) have a strong inter-correlation, which is consistent with their representation of essentially the same information in different formats. Additionally, "data", which is the payload size parameter, has high correlations with several signal metrics, such as "lsnr" (LSNR), "datr" (data rate") and "rssi" (RSSI), which is expected according to the fundamental principles of LoRaWAN functionality, where the payload size can influence and be influenced by several transmission characteristics. They still indicated that "tmst" is very isolated from the other dataset parameters, which shows how this parameter does not have a strong direct relationship with other measured variables, which might indicate that the network behaviour is consistent over time or it's necessary more complex temporal analysis in future work.

When it comes to values distributions and numerical ranges of some of the dataset's most relevant parameters, Esteves et al. (2024) showed that the most common configuration uses SF of 12 and BW of 125, which happens when the Adaptive Data Rate bit is active in frame control. The payload length usually is around 70 bytes, being in a numerical range from 20 to 418 in RXPK messages, and 24 to 102 in TXPK

messages. The “size” parameter is in the numerical range of 10 to 209 in RXPK messages, and in the numerical range of 12 to 51 in TXPK messages. This suggests that RXPK messages usually contain more data than TXPK messages. RSSI values are notably concentrated around -118 dBm, being in a numerical range of -125 dBm to -71 dBm, values of LSNR concentrate around -15dBm, being in a numerical range of -24dBm to 15dBm. The parameter “chan” has values between 0 and 24 in RXPK messages, and “freq” assume values between 867.1 and 868.5 in RXPK messages and between 867.1 and 869.525 in TXPK messages.

All files have the name pattern “<type>_<date>.log”, where “type” is “rxpk” for files with RXPK messages (messages that the gateway sends directly to the NS) or “txpk” for files with TXPK messages (messages that the NS sends directly to the gateway), and “date” was the day when the LoRaWAN messages from the file were captured, in YYYYMMDD format (for example, “rxpk_20200603.log” contains all RXPK messages captured in June 3rd, 2020). Because of this name pattern from the source datasets, the binding of the dataset was easy to do with Python code.

The datasets used by the models result from binding this set of files in two datasets: one for all JSON files with RXPK messages (whose name starts with “rxpk”), and other for all JSON files with TXPK messages (whose name starts with “txpk”).

In the IDS implementation available in the GitHub repository published by Tavares (2025), the .log datasets are available in a subdirectory named “datasets”, and after you run the indicated python script according to the provided README file, you can generate the datasets with some pre-processing techniques already applied (techniques from section 3.3.2.1 to section 3.3.2.6).

3.3.2 Data Pre-Processing

This phase is crucial for training of ML models, since it consists on applying modifications to the input dataset that improve the efficiency and efficacy of these models on training and classification of new instances. The applied pre-processing techniques were Feature Selection (section 3.3.2.1), Feature Extraction (section 3.3.2.2), dataset filtering for the specific device and type of messages with missing values imputation and inserting manual intrusions included (section 3.3.2.3), Feature Assembling (section 3.3.2.4), Feature Scaling (section 3.3.2.5) and Feature Reduction (section 3.3.2.6).

3.3.2.1 Feature Selection

One of the aspects that most undermines the effectiveness of ML models is the “curse of dimensionality”. The dimensionality of a dataset consists of its quantity of different attributes, represented by its columns. An higher dataset dimensionality leads to worse results in training and testing of ML models. Therefore, to improve the model’s efficiency and efficacy, a great and common practice is to reduce the number of attributes in the

dataset, removing irrelevant and correlated attributes. This helps the model reduce the risk of false alarms, as explained by Mohammed et al. (2024), but most importantly to avoid false negatives, which are anomalous messages classified as normal messages. With this, the level of accuracy, security and reliability of the ML models is enhanced. From the datasets whose structure is specified in Appendix B, specifically on sections B.5.1 and B.5.2, relevant attributes are some of the parameters coming from the LoRa gateway, such as RSSI, LSNR and tmst, also specified in the GitHub LoRa-net repository published by Lora-net (2013), and also some parameters derived from the LoRaWAN message payload, such as DevEUI, DevAddr, MIC and DevNonce, which are also specified in the LoRaWAN specification published by LoRa Alliance (2018). Therefore, the attributes which didn't follow these requirements were removed. In RXPK and TXPK messages, the following attributes were removed from static datasets:

- **Irrelevant attributes**
    - **Fromip, ip and port:** the source IP address and port are irrelevant for anomaly detection;
    - **Direction:** it's always "up";
    - **NetID:** knowing the network identifier will not indicate anomalies in the message content itself;
    - **Modu:** it's always LoRa;
    - **FRMPayload:** we only need the size if the frame payload, and the payload itself can be too big to be supported by PySpark when converted from hexadecimal to decimal;
    - **Stat:** the CRC status is irrelevant;
    - **Type:** the type of message is already known by the message structure itself, it's not necessary to detect intrusions;
    - **Totalrxpk in RXPK:** to detect intrusions, the number of RXPK messages in each LoRaWAN packet isn't relevant;
    - **Attributes not available in the specifications;**
- **Missing attributes**
    - **CFList in RXPK:** only contains NULL values in RXPK messages, because this parameter is only available in JA messages;
- **Redundant and correlated attributes**
    - **Recv_date, time, tmms:** these are all time-related parameters, and it's only necessary one parameter to indicate the time when the message was sent, and tmst was chosen since this is the internal timestamp calculated by the gateway, and these three time-related parameters are equivalent but are dependent on the timezone;

- **FreqCh4, FreqCh5, FreqCh6, FreqCh7 and FreqCh8 in TXPK:** these are all derived from CFList;
- **DLSettingsRX1DRoffset and DLSettingsRX2DataRate:** these parameters are inside DLSettings, so only DLSettings is used to have less attributes in the dataset (one instead of two);
- **FCtrlACK and FCtrlADR:** these parameters are inside FCtrl, so only FCtrl is used to have less attributes in the dataset (one instead of two);
- **FHDR:** it results from the division of DevAddr, FCtrl, FCnt and FOpts which were maintained in the dataset because these are relevant parameters for intrusion detection, and each one of them is used for different purposes in the context of anomaly detection. Moreover, FHDR maximum size is more than 15 bytes, so after being converted to decimal, it results in a maximum size of more than 38 digits, which is the PySpark size limit. So, by saving FHDR after converting to decimal, PySpark would throw an error;
- **MessageType:** it's already inside MHDR;
- **RxDelayDel:** it's already inside RxDelay;
- **Data:** it's equivalent to PHYPayload but encoded with Base64.

3.3.2.2 Feature Extraction

Some relevant features were transformed to an adequate format and others were just used to extract some parameters inside them. Most attributes were initially strings, but ML models use a type of column which contains a vector with all the assembled values of the attributes, and they only support numerical values on these vectors. Therefore, it was necessary to guarantee that all attributes would be numerical. To do so, the following steps were performed:

- SF and BW were extracted from datr and cast to integer. Then datr was removed from the dataset;
- PHYPayloadLen was created, corresponding to the length of PHYPayload. Then PHYPayload was removed from the dataset;
- CFListType was created, coming from the last byte of CFList, and then the last byte of CFList was removed from the CFList parameter itself. This ensures that CFList is not too large for PySpark after being converted from hexadecimal to decimal;
- On RXPK messages, chan and lsnr inside rsig array were extracted and aggregated with chan and lsnr outside rsig array but also inside rxpk array, to form two arrays: chan and lsnr, each one with the two values of the corresponding parameter. Then, four attributes were created: chan1, the first element of the chan array, chan2, the second element of the chan array, lsnr1, the first element

of the lsnr array and lsnr2, the first element of the lsnr array. After that, the arrays rsig, chan and lsnr were removed from the dataset;

- Codr was converted from string to float, because it's a fraction;
- The following attributes were converted from hexadecimals in strings to decimal values as integers: AppEUI, DevEUI, DevNonce (these three only in RXPK messages), AppNonce, DLSettings, FPort, MIC, FCtrl, FCnt, FOpts, MHDR, CFList, CFListType, RxDelay (in both RXPK and TXPK messages);
- An attribute called "intrusion" was created to set as 1 in packets where manual intrusions are introduced to test the model, and then compute evaluation metrics to determine the model efficacy on testing, being used as a "label" during testing.

On the other hand, RXPK messages coming from LoRa gateway in real time only come with the following parameters inside "rxpk" array: chan, codr, data, datr, freq, lsnr, modu, rfch, rssi, size, stat, time and tmst. They don't come with LoRaWAN parameters previously extracted, so it was necessary to perform different pre-processing steps in this case. These were:

- Extract all parameters from "message" attribute that contains the "rxpk" array;
- Remove "modu", "stat" and "time" from the dataset, since these are irrelevant attributes;
- Extract all LoRaWAN relevant parameters from "data" attribute:
  - Compute PHYPayload by decoding "data" using Base64;
  - From PHYPayload, compute AppEUI, AppNonce, DLSettings, CFList, CFListType, DevAddr, DevEUI, DevNonce, FCnt, FCtrl, FOpts, FPort, PHYPayloadLen, MIC, MHDR and RxDelay, according to the specification published by LoRa Alliance (2018b), and the steps described on this section to transform some of these attributes;
- Remove "data" and "PHYPayload" from the dataset after extracting the relevant parameters from these attributes.

3.3.2.3 Preparing Dataset to Create Model

In the IDS, a model is created for each different DevAddr and type of message (RXPK or TXPK), which means that there are, at most, two created models for each device that has joined the network. The input datasets are the datasets that were generated from the provided datasets, after applying the pre-processing steps described from sections 3.3.2.1 to 3.3.2.2.

The first step to prepare the dataset to create the model was to filter the dataset to contain only the messages from the corresponding DevAddr and type of message. The second step is to verify if the filtered dataset has enough samples to create the model, according to the defined limit, which is 50 samples if the IDS is executed only to create

ML models (Figure 3.3), or 30000 if the IDS is receiving messages from the LoRa gateway in real time (Figure 3.2), since the gateway can send repeated messages and if the model is created with too few messages, the model might not fit well to the normal traffic patterns of the device and it can consider new normal messages as anomalous. If the training dataset size is below the defined limit, the model will not be created, but if the dataset has at least 1 sample, it will still be used in real-time classification to bind with the existing device samples for the indicated type. If the dataset size is equal or greater to the defined limit, the model will be created, and the following pre-processing steps will be performed to prepare dataset for model training:

- Columns that only contain missing values in a dataset or sub-dataset are removed.
- In the remaining columns, the missing values are replaced by the mean of the corresponding attribute. To do so, it was necessary to ensure that these columns only contain numeric values, and for that reason, categorical values were whether removed or transformed into numerical values, as explained in section 3.3.2.2.

If the IDS is executed in the context of model creation without real-time message processing and classification, the dataset is split into training (80% of samples) and testing (20% of samples). After that, manual intrusions are introduced on test dataset, and that is done by modifying all the relevant attributes, that were not removed on feature selection or that were created on feature extraction, to assume abnormal values that the model hasn't learnt during training. For that, in SF and BW, values that are not present in training are introduced in some samples to indicate intrusions. For example, SF values can rely between 7 and 12, as specified by Semtech. But if, in the device dataset, SF only is 7, in test dataset some samples are modified for SF to assume values between 8 and 12, and the model is expected to detect those samples as intrusions. In case of parameters such as size and PHYPayloadLen that are based on data length, to introduce intrusions, very large values are inserted.

If the execution of the IDS includes real-time message processing and classification, the whole device dataset will be used for model training, and testing is not applied.

3.3.2.4 Feature Assembling

For training and testing on PySpark, ML algorithms use a specific vector column which contains all values that are used for that. That column is creating by using a PySpark function called VectorAssembler, which is a feature transformer that merges multiple columns into a vector column (a DenseVector column). Therefore, to train the model with the input dataset, that vector column results in merging the values of the rows of all columns representing all dataset relevant attributes, except "DevAddr" and "intrusion", after applying, on that dataset, all FS and FE techniques described on sections 3.3.2.1 and 3.3.2.2, respectively. All these techniques, as described on these two sections,

ensure that the values of all relevant attributes that are still on the dataset are all numeric, which is crucial because for training and testing, ML algorithms only support numerical values inside the vector they use.

3.3.2.5 Feature Scaling

Feature scaling is a crucial pre-processing step which significantly improves the performance and reliability of ML models during both training and testing. To do so, in this project, a PySpark function called StandardScaler was employed to transform all features to have zero mean ("withMean" = True) and unit variance ("withStd" = True).

Centring the data to zero mean ensures that features with larger absolute values don't disproportionately influence the learning process. For example, the feature PHYPayloadLen typically assumes only positive values, and, without this transformation, it could be interpreted as more important than RSSI by some ML algorithms, because RSSI typically assumes negative values. Without centring data to an equal mean, some ML algorithms could incorrectly prioritize features with larger absolute values such as PHYPayloadLen simply because of its higher numeric scale, even if they are not more relevant. Instead, by centring all data to zero mean, all features have the same numerical scale, which improves the results of ML algorithms that are sensitive to the features' scale because the probability of detecting anomalies is the same on all attributes, and it's no longer biased by differing mean values.

In addition, scaling to unit variance is essential to prevent features with higher variability from dominating the model's optimization process. For example, PHYPayloadLen tends to have a significantly higher variance than SF, which typically assumes values only between 7 and 12. Without variance normalization, some ML algorithms tend to become more sensitive to variations in attributes with higher variance like PHYPayloadLen than attributes with lower variance like SF, which could result in missed anomalies on attributes with lower variance for ignoring subtle but meaningful deviations on these. Therefore, by standardizing values to unit variance, the model is equally sensitive to deviations across all attributes, regardless of their original variance.

By applying standardization with zero mean and unit variance, all features contribute equally in terms of scale, preventing the model from being biased towards any attribute due to its numerical range or variability. This normalization is specially fundamental to get adequate results on distance-based algorithms such as LOF (section 2.4.2.2), and algorithms sensitive to scale such as PCA (section 3.3.2.6.1).

3.3.2.6 Feature Reduction

FR consists of transforming the dataset into a smaller representation with lower dimensionality, but maximizing the dataset variance, to improve the ML models' efficacy and efficiency in training and classification of new instances. On this section, three

techniques are approached: PCA (section 3.3.2.6.1), SVD (section 3.3.2.6.2) and Autoencoders (section 3.3.2.6.3). After that, the chosen techniques to apply on this project are specified on section 3.3.2.6.4.

3.3.2.6.1 Principal Component Analysis

PCA is a FR technique which consists of identifying the principal components or directions along which data variance is maximized. It's an effective technique to project high-dimensional dataset like the dataset used on this project, into a lower-dimensional space, which allows PCA to capture essential features while reducing noise. It finds orthogonal directions called principal components, that capture the maximum variance in the data, as explained by Zaheer et al. (2025) and Jolliffe & Cadima (2016). These principal components are uncorrelated variables that successfully maximize dataset variance and find these involves solving an eigenvalue / eigenvector problem. PCA involves eigenvalue decomposition of the covariance matrix of the data according to the equation (3.1).

$$C = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})^T \tag{3.1}$$

C is the covariance matrix, x_i represents the data points and $\bar{x}$ is the mean of the dataset. The eigenvectors of C represent the principal components, and their corresponding eigenvalues indicate the variance explained by each component. Recent studies have proven that PCA has potential to significantly improve the performance of ML models. However, PCA has a disadvantage: it assumes that data is linearly separable and it's sensitive to scaling, which adds the need to perform feature scaling before applying PCA. Feature scaling was previously explained in section 3.3.2.5.

3.3.2.6.2 Singular Value Decomposition

SVD is a FR technique which consists on simplifying the dataset representation by decomposing a matrix X into singular values, which consist on three components, as described on the equation (3.2) and also by Zaheer et al. (2025).

$$X = UDV^T \tag{3.2}$$

As also explained by Thi (2023), D is a diagonal matrix containing singular values which represent the relative contribution rate of each component for the original data variability and are non-negative values in descendent order. U and V are orthogonal matrices, and while U represents the left singular vectors of X, V represents the right singular vectors of X. V^T is the transpose of V, and it represents the main directions on the space of the original matrix. V^T transforms the data from the original space into a new system of coordinates, where directions are orthogonal and sorted by relevance, according to the singular values in D.

This FR technique is highly effective in dimensionality reduction while preserving the most significant singular values, which correspond to the strongest patterns in the data.

By testing the solution, it was possible to find out that this technique, like PCA, has a much better performance when scaling is previously applied to the features used.

3.3.2.6.3 Autoencoders

AE is an unsupervised DL algorithm that, as previously explained in section 2.4.2.11, attempts to reproduce the input dataset with minimal reconstruction error using an encoder function and a decoder function. This makes this algorithm useful for FR, but computationally intensive. Also, on this project's implementation, the training and testing dataset are different due to the intrusions manually inserted on the test dataset, which might complicate AE learning rather than improving its generalization.

3.3.2.6.4 Final Choice of Techniques

The two chosen techniques to test the IDS and compare the different algorithms were PCA and SVD. This choice was made because these techniques are simpler to implement, since they provide native support for PySpark unlike AE, and also because these are computationally less expensive than AE and have much more effective results. On IDS testing, some ML algorithms had better results when applying PCA and others had better results when applying SVD, and in the end a comparison of all algorithms' results was done considering previous application of Feature Scaling and SVD, because SVD can capture more data variance with less components, which means lower dataset dimensionality. That's more detailed in Chapter 4.

In the project implementation, the idea is to find the optimal number of PCA / SVD components which allows to capture, at least, 99% of the input dataset variance. This ensures that the resulting features vector, which is later used by ML models for training and testing, has the lowest dimensionality possible.

3.3.3 Data Processing

After applying all pre-processing steps described on section 3.3.2, the next step was data processing. This involves model training and testing. As previously explained, when the IDS is receiving new messages in real time, it only applies model training, not testing, following the architecture described on Figure 3.2. When IDS is only being executed to create models without receiving messages in real time from a LoRa gateway, the model goes through training and testing, following the architecture described on Figure 3.3.

3.3.3.1 Model Training

In this phase, the model is trained using the training dataset, which consists of samples where at least most of the attribute values are assumed to represent normal or expected traffic behaviour. Training allows the model to learn the underlying statistical patterns or

distributions which characterise normal behaviour. By doing so, the model builds a baseline of what is considered typical expected behaviour within the system.
The training process is especially important in anomaly detection scenarios, where it's assumed that the training data contains little to zero anomalous samples. As a result, the model learns to characterize what is normal. After training, when the model is applied to the test dataset or to new unseen data, it can identify anomalies by detecting deviations from the learnt patterns. Therefore, model training serves as the foundation for the detection of behavioural deviations in new data. When new messages are received in real time from the LoRa gateway, the model is re-trained with those messages if they are classified as normal by that model.

3.3.3.2 Model Testing

After training, the model is evaluated using new data that wasn't seen during the training phase. The objective of model testing is to assess the model's ability to generalize and effectively detect anomalies in unseen samples. If a new sample significantly deviates from the patterns learnt during training, the model should classify it as an anomalous instance.
Although the model is trained in an unsupervised learning context, the test dataset includes an hardcoded label that, as previously explained in section 3.3.2.2, is set to 1 in the test samples where manual intrusions are introduced, and set to 0 in the remaining test samples. This label is exclusively used to evaluate the ML model performance on testing by computing the evaluation metrics that are used for it, since an unsupervised approach is being employed, as previously explained. The most used evaluation metrics in ML are confusion matrix, accuracy, precision, recall and F1-score. Table 3.2 shows the confusion matrix, equation (3.3) shows the expression of accuracy, equation (3.4) shows the expression of precision in class 1 (anomaly), equation (3.5) shows the expression of precision in class 0 (normal), equation (3.6) shows the expression of recall in class 1 (anomaly), equation (3.7) shows the expression of recall in class 0 (normal) and, finally, equation (3.8) shows the expression of F1-Score.

Table 3.2 - Confusion Matrix Structure

Predicted / Real	0	1
0	TN	FN
1	FP	TP

$$accuracy = \frac{TP+TN}{TP+FP+TN+FN} \tag{3.3}$$

$$precision\ (class\ 1) = \frac{TP}{(TP+FP)} \tag{3.4}$$

$$precision\ (class\ 0) = \frac{TN}{(TN+FN)} \tag{3.5}$$

$$recall\ (class\ 1) = \frac{TP}{(TP+FN)} \tag{3.6}$$

$$recall\ (class\ 0) = \frac{TN}{(TN+FP)} \tag{3.7}$$

$$F1Score\ (class\ X) = 2 * \frac{Precision(class\ X) * Recall(class\ X)}{Precision(class\ X) + Recall(class\ X)} \tag{3.8}$$

The confusion matrix shows the distribution of predicted and actual classifications. It's used to evaluate how many instances were correctly classified, containing the number of instances considered TN, the number of instances considered TP, the number of instances considered FP, and the number of instances considered FN. TN are normal instances correctly classified as normal instances, TP are anomalous instances correctly classified as anomalous instances, FP are normal instances incorrectly classified as anomalous instances and FN are anomalous instances incorrectly classified as normal instances. Accuracy measures the overall rate of correctly classified instances. Precision indicates the proportion of instances predicted as a certain class that were of that class. In case of precision of class 1, it indicates the proportion of predicted anomalies that are real anomalies, according to the real label. Recall indicates the proportion of messages labelled as a specific class, that are correctly classified by the model in that class. For example, recall of class 1 indicates the proportion of actual anomalies (labelled as 1) that are correctly identified as anomalies. F1-Score is a metric which provides the balance of precision and recall from the corresponding class.

All these evaluation metrics have their importance on model performance evaluation on testing. However, in anomaly detection, the most relevant and critical evaluation metric is recall from the class 1, as it directly reflects the model's ability to correctly detect anomalous messages. What must be avoided the most are FN, which are anomalous messages classified as normal messages, since this represents lost anomalies, which represents a threat in IDS reliability and security. To ensure the minimization of FN, precision of class 0 could also be considered another alternative to measure the model capability of detecting anomalies, even though recall from class 1 is sufficient for this. In the other hand, FP may indicate test samples that were labelled as normal but possibly contain previously unknown anomalies that were not manually injected, given that only a subset of anomalies was manually injected into the test set. However, it's also important to ensure that there are not too many false alarms, with some of them being useless, which might happen when there is too much concern on only optimizing the recall from class 1 by increasing too much the training outlier rate as training parameter of the implemented ML algorithms. Therefore F1-Score is also an important metric for anomaly detection in this solution, being probably the second most important metric on this solution. Accuracy is also a relevant metric for this solution since it represents the overall correctness of the model, but it's not as relevant as recall and F1-score because,

as explained by Esteves et al. (2024), it can be misleading in cases of class imbalance - which happens in most cases in this solution.
To evaluate the IDS, multiple ML algorithms described in section 2.4.2 were implemented and tested under the same conditions. The results of these experiments are presented and discussed in Chapter 4.

3.3.4 Stream Processing Configuration Steps

The IDS was configured to run on a virtual machine with 64 GB of RAM memory, 8 CPUs and the version 24.04.2 LTS of Ubuntu, a popular Linux distribution. To allow the virtual machine where the IDS is running to receive LoRaWAN messages in real time, it was necessary to perform additional steps. First, it was necessary to configure a LoRa gateway that would send periodically LoRaWAN messages, ensure that those messages would be forwarded to the VM where the IDS is running and then configure Apache Spark on the IDS to receive those messages from the VM where IDS is running.
LoRaWAN messages come from a LoRa GW which was previously provided, and that gateway was configured to forward those messages to the machine where the IDS is running. It was necessary to apply extra configurations to forward those messages from the LoRa gateway to the Spark application running inside the IDS, and that was done following the architecture illustrated on Figure 3.4. More details of these configuration steps are detailed on the Appendix A.

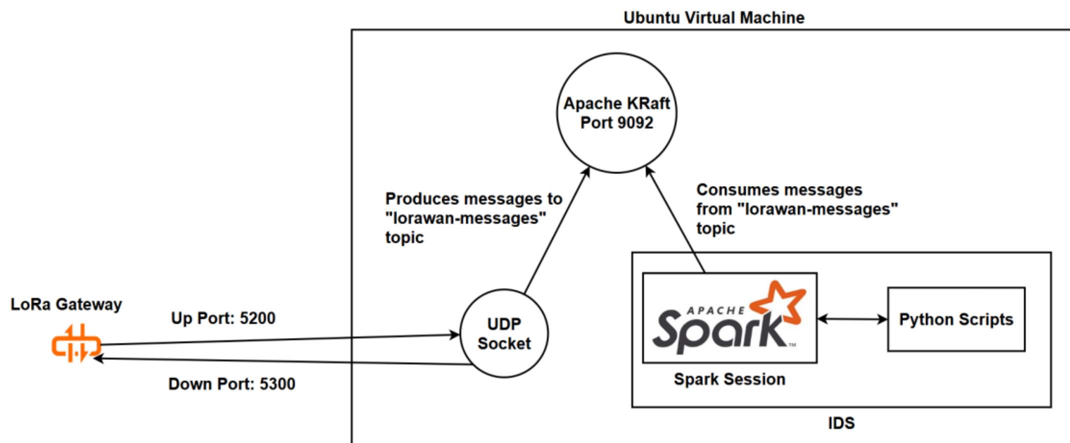

Figure 3.4 - Stream Processing Setup

An UDP socket was configured within the IDS, using Python, out of the Spark context, to receive traffic from the LoRa gateway. After that, it was necessary to configure an intermediary system which receives that traffic from the UDP socket, and then forward it to Spark Structured Streaming, enabling real-time message processing. The LoRa gateway does not natively support secure protocols such as Kafka or TCP, which are compatible by Spark. Conversely, Spark does not support clear-text protocols like UDP, making it impossible to establish a direct connection between the LoRa gateway and the

Spark application. As a result, it was necessary to configure this intermediary. To do so, two potential intermediaries were considered: a Kafka server and a TCP Socket.
While TCP sockets are simpler to implement, they lack fault tolerance. Kafka, on the other hand, requires a more complex setup, because Apache KRaft, previously explained in section 3.1.2, must be locally installed in the VM where IDS is running. After that, it's necessary to setup and initialize Apache KRaft before running Spark, since Spark will connect to Apache KRaft to consume the messages that come from the UDP Socket to the Kafka topic "lorawan-messages", from which Spark will subscribe to be able to consume the messages. Despite the additional complexity, both options would be valid for this setup, but after testing both solutions on the implementation, Kafka was the chosen option, having as advantages its fault tolerance and its ability to temporarily store messages even when Spark isn't running.
The last step in the stream processing setup involved configuring Spark Structured Streaming to consume the messages sent to Kafka. Spark was configured to establish a connection to Apache KRaft on port 9092 and to subscribe to a created Kafka topic called "lorawan-messages" to consume the LoRaWAN messages in that topic.

3.4 Used Tools: Frameworks and Libraries

This section describes all tools used to implement this project. The frameworks used to build and orchestrate the core components of the system are specified in section 3.4.1. The Python libraries used to support ML, data processing and anomaly detection are specified in section 3.4.2.

3.4.1 Frameworks

3.4.1.1 MLFlow

MLFlow is an open-source platform used to manage the lifecycle of ML experiments. This is a tool which is very well-integrated with Spark, and it was used especially to save ML models after training them, storing them as artifacts to be later retrieved to classify new instances or replace them with new re-trained versions of those models, but also to log the associated pre-processing models (PCA and SVD models) and evaluation metrics such as confusion matrix, accuracy and recall, and record parameters as tags associated with those models, namely the DevAddr and the type of message (TXPK or RXPK). This facilitates the usability and reusability of ML models during the execution of the IDS while receiving LoRaWAN messages from the LoRa gateway.

3.4.1.2 PySpark

PySpark is the Python API for Apache Spark, a powerful distributed data processing framework previously explained on section 3.1.3. It was used to build the real-time anomaly detection pipeline using Spark Structured Streaming (section 3.1.3.2), enabling real-time data processing in a scalable and fault-tolerant way, and also to develop all the necessary code to implement the solution, including steps such as large-scale data processing, data pre-processing and development of ML algorithms.

3.4.2 Libraries

3.4.2.1 Scikit-Learn

Scikit-learn is a popular Python library for ML. Even though PySpark provides native support for many different ML algorithms, it does not provide native support for many ML algorithms implemented and tested on this project. In this project, scikit-learn was used to implement LOF (section 2.4.2.2), Isolation Forest (section 2.4.2.6) and One-Class SVM (section 2.4.2.8). These algorithms are natively supported by scikit-learn but they are not natively supported by PySpark (section 3.4.1.2).

3.4.2.2 NumPy

NumPy is the fundamental package for numerical computing in Python. It was used in this project to perform array operations, to ensure that the vectors containing the dataset assembled features would have a compatible format to be used by models not created by PySpark, namely scikit-learn (section 3.4.2.1) and Pyod (section 3.4.2.3) models, and also to save and load SVD matrices, during stream processing, which are used to apply pre-processing on new instances where the corresponding ML model relies on SVD to apply FR to transform the data before being prepared for classification.

3.4.2.3 Pyod

Pyod is a Python library used to detect anomalies in multivariate data. It was solely used in this project to implement HBOS, since this is not natively supported by PySpark (section 3.4.1.2).

4 Evaluation and Results

By developing Python scripts to create, train and test the ML models to develop this solution, it was possible to test the efficacy and efficiency of these models, created according to the architecture on Figure 3.3, to evaluate their ability to detect anomalous messages in the provided static datasets. On this chapter, the chosen ML algorithms to test the IDS will be covered, detailing the results of each algorithm on section 4.1. Then, on section 4.2, a comparation of these implemented algorithms with each other is specified, showing which algorithm delivered the best results on testing, and a final discussion about the experiments that were performed throughout the whole development of the solution is described on section 4.3.

4.1 Results

To show the results, 8 devices were chosen according to their DevAddr: 26012130, 260124F1, 26012917, 260125E6, 26012017, 0000C056, 0000B516, 0000B545. These devices were chosen for testing to show the evolution of the testing results while increasing the number of training samples. The figures from section 4.1.2 to section 4.1.6 and in section 4.2 contain the test results, including the most relevant metrics and the time of the model generation for each case. Note that the results of the models illustrated in these sections aren't deterministic, as these can change if the models for the same devices are created and tested several times. Even though a static integer value is used as seed for reproducibility, this happens because most models rely on computed distributions and also computations of decision functions to decide if a new data point will be considered an anomaly or not. However, it's expected to minimize the FN rate in order to detect as much anomalies as possible, whether manually inserted or just present on the test dataset even if labelled as normal messages, but also not to trigger too much false alarms (FP). It's also expected that the model improves its results in testing while increasing the number of training samples. To achieve these results, the number of synthetic anomalies that were injected on the test dataset followed the equation (4.1).

$$num_intrusions = \max(1, \min(round(0.2 * N), 30)) \quad (4.1)$$

This dynamic number of intrusions allowed an optimization of the results of the ML algorithms, ensuring a maximization of recall but also trying to avoid too much false alarms.

4.1.1 Devices Used for Testing

Before moving on to the results of each algorithms, the total number of samples for each device and each message type, and the time taken for pre-processing of the corresponding model, are detailed on Table 4.1. The last 3 devices (with DevAddr of 0000C056, 0000B516 and 0000B545) don't have TXPK samples, but they have a considerably high number of RXPK samples, which is relevant to prove that the model is relatively more efficient on testing after being trained with a lot of data. Note that the number of samples shown on Table 4.1 is not the number of training samples, since the training samples are 80% of all samples of each device, and 20% of the total of device samples goes for testing, as previously explained in section 3.3.2.3. In the provided datasets, the device 0000B545 is the one with the most samples. Time of pre-processing assumes application of Feature Scaling and SVD.

Table 4.1 - Number of device samples and time of pre-processing

DevAddr + Type	Number of Samples	Time of pre-processing
26012130; RXPK	141	11 s
26012130; TXPK	50	3 s
260124F1; RXPK	366	4 s
260124F1; TXPK	100	3 s
26012917; RXPK	713	4 s
26012917; TXPK	492	3 s
260125E6; RXPK	1504	4 s
260125E6; TXPK	844	3 s
26012017; RXPK	3575	4 s
26012017; TXPK	2350	3 s
0000C056; RXPK	8857	5 s
0000B516; RXPK	20314	4 s
0000B545; RXPK	36975	4 s

The Table 4.1 shows that, in the first generated model (device 26012130, in RXPK messages), the pre-processing takes more time (11 seconds) than in the next models, which is an behaviour expected by Apache Spark because, as previously explained in section 3.1.3, Apache Spark is based on memory computations and it adopts a lazy evaluation approach, where transformations are only executed where an action is called, which makes the first operation to involve the preparation and optimization of the whole execution plan. Naturally, the pre-processing should take slightly more time on larger datasets, but the Spark architecture mainly focused on batch processing helps to

significantly increase the static dataset processing efficiency, for large and small datasets.

4.1.2 Isolation Forest (SkLearn Implementation)

Figure 4.1 to Figure 4.5 show the results of the implemented sklearn version of Isolation Forest, without normalization and with feature scaling, and also comparing PCA with SVD with feature scaling included.

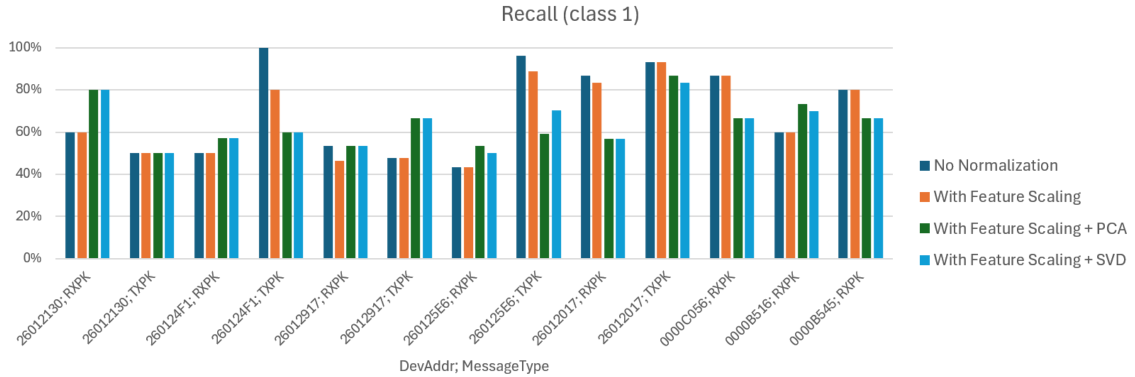

Figure 4.1 - Recall (class 1) for sklearn-based Isolation Forest

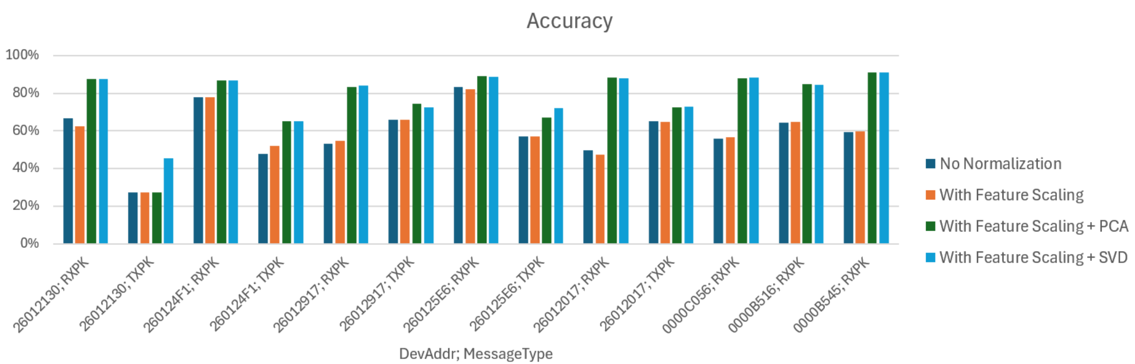

Figure 4.2 - Accuracy for sklearn-based Isolation Forest

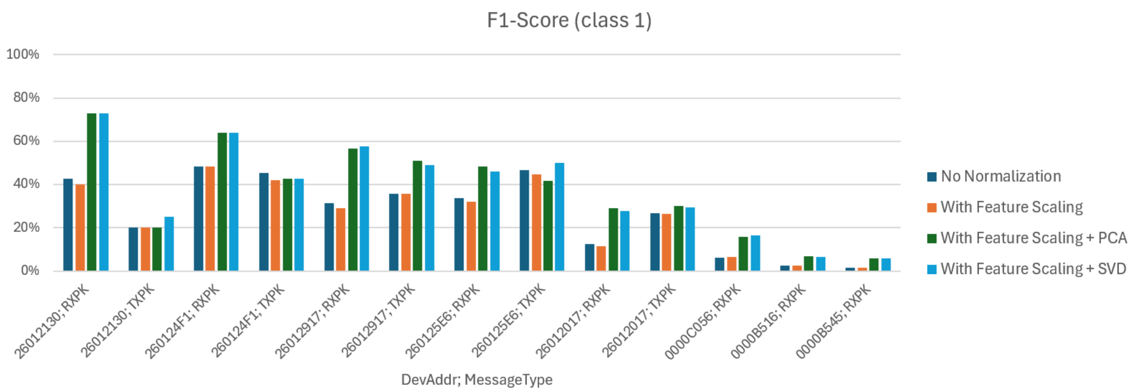

Figure 4.3 - F1-Score (class 1) for sklearn-based Isolation Forest

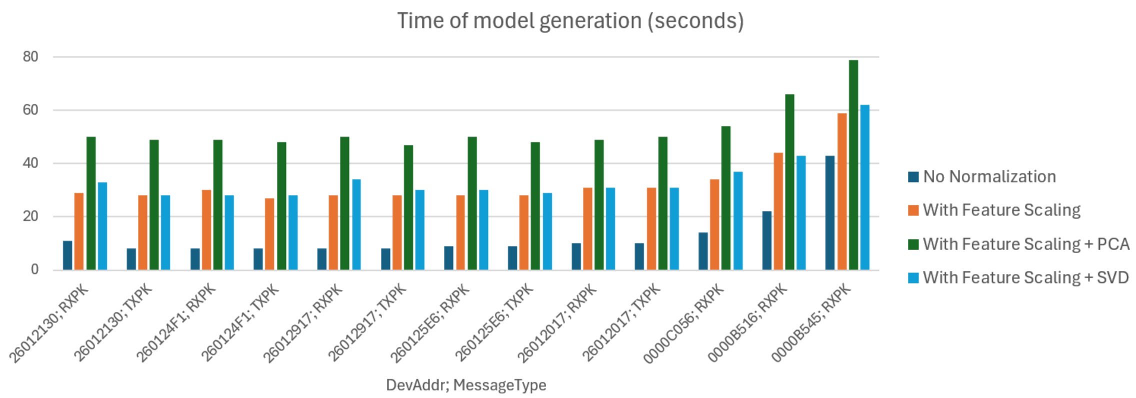

Figure 4.4 - Time of model generation for sklearn-based Isolation Forest

Normalization	Values \ DevAddr + Type	26012130; RXPK	26012130; TXPK	260124F1; RXPK	260124F1; TXPK	26012917; RXPK	26012917; TXPK	260125E6; RXPK	260125E6; TXPK	26012017; RXPK	26012017; TXPK	0000C056; RXPK	0000B516; RXPK	0000B545; RXPK
No normalization	Predicted Positives	9	8	15	17	67	35	47	84	385	180	793	1410	2951
	Predicted Negatives	15	3	53	6	72	71	260	53	336	262	961	2540	4256
With Feature Scaling	Predicted Positives	10	8	15	14	61	35	51	80	399	181	780	1393	2915
	Predicted Negatives	14	3	53	9	78	71	256	57	322	261	974	2557	4292
With Feature Scaling + PCA	Predicted Positives	6	8	11	9	25	34	36	50	87	143	222	617	645
	Predicted Negatives	18	3	57	14	114	72	271	87	634	299	1532	3333	6562
With Feature Scaling + SVD	Predicted Positives	6	6	11	9	24	36	35	49	92	139	212	623	655
	Predicted Negatives	18	5	57	14	115	70	272	88	629	303	1542	3327	6552

Figure 4.5 - Number of predicted positives and negatives for sklearn-based Isolation Forest

This algorithm is configured with a dynamic training outlier rate, used to train the model, which allows the value of the corresponding parameter to be adjusted according to the nature of the training dataset. More details of the training parameters configuration are specified on the Appendix B.2. The results from Figure 4.1 to Figure 4.5 show that even without feature scaling and feature reduction, the algorithm does not have bad results at all. In some devices, there is an improvement after applying feature reduction, but in other devices the best results are achieved without FR. Comparing PCA and SVD, in most of the devices, both techniques result in a similar performance, where SVD always results in a faster model generation. Without feature scaling and reduction, the model is faster to be generated, and PCA is the technique that makes the model to take the longest time to be generated.

4.1.3 Isolation Forest (Custom Implementation)

In the project's implementation, an IF implementation, which involves Apache Spark operations, has been tested, based on the implementation in the GitHub repository published by Verbus et al. (2019). Its results are presented from Figure 4.6 to Figure 4.10, without normalization and with feature scaling, and also comparing PCA with SVD with scaling included.

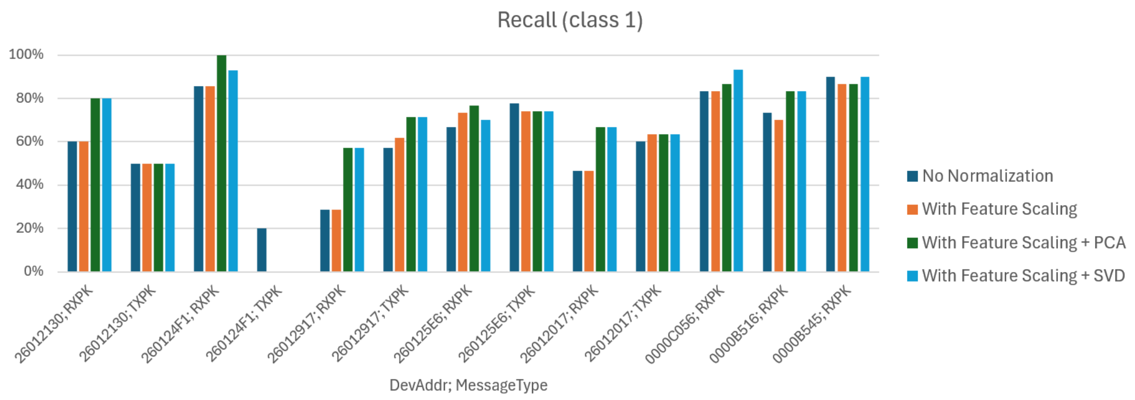

Figure 4.6 - Recall (class 1) for custom version of Isolation Forest

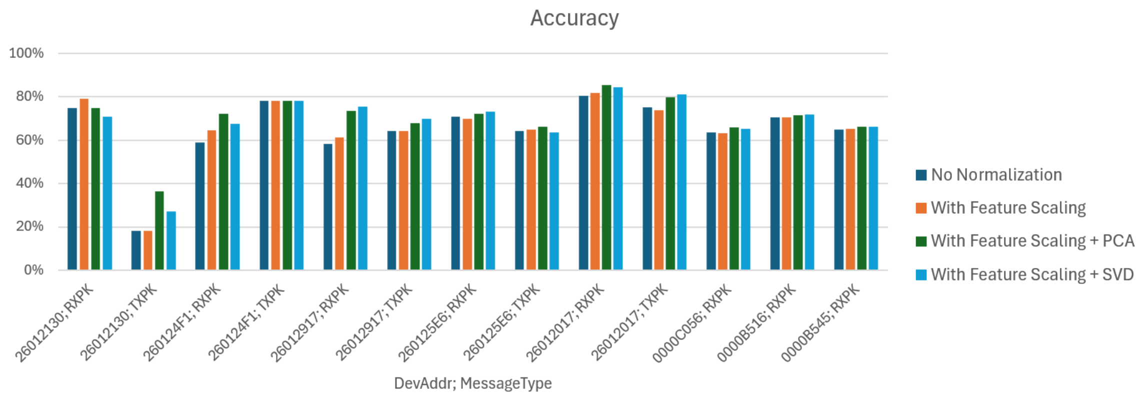

Figure 4.7 - Accuracy for custom version of Isolation Forest

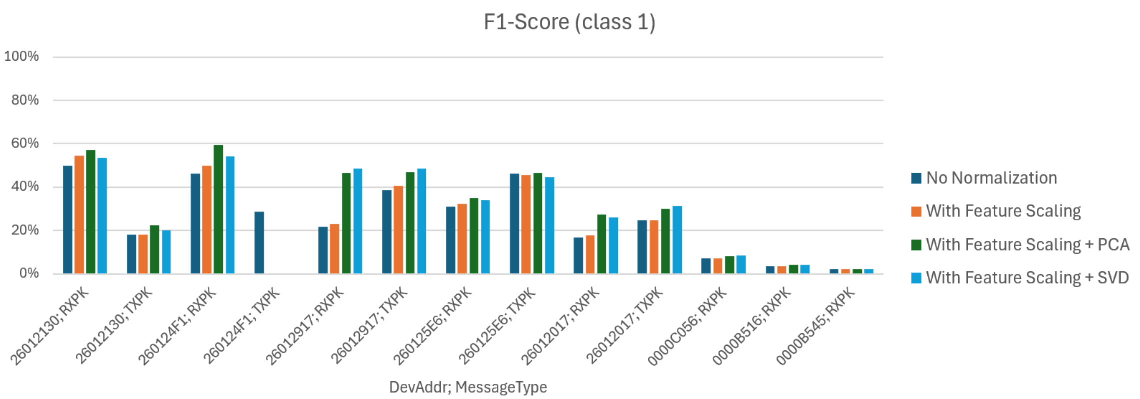

Figure 4.8 - F1-Score (class 1) for custom version of Isolation Forest

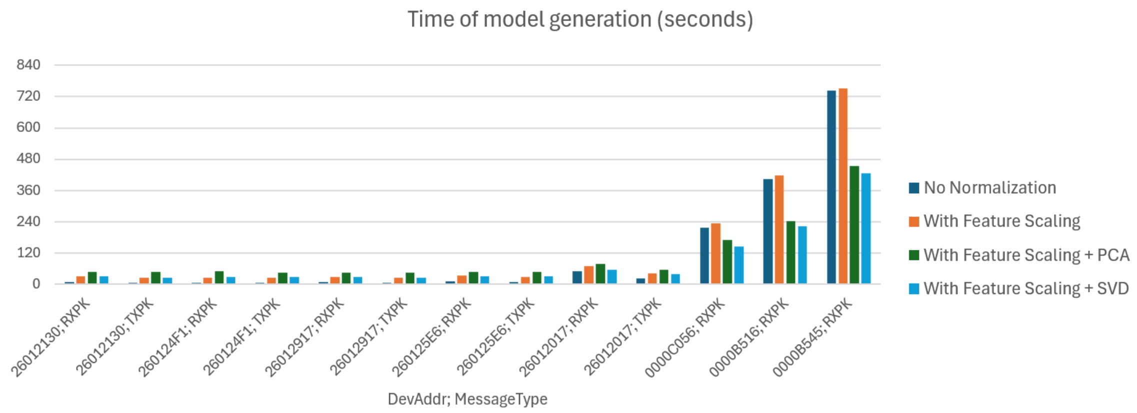

Figure 4.9 - Time of model generation for custom version of Isolation Forest

Normalization	DevAddr + Type / Values	26012130; RXPK	26012130; TXPK	260124F1; RXPK	260124F1; TXPK	26012917; RXPK	26012917; TXPK	260125E6; RXPK	260125E6; TXPK	26012017; RXPK	26012017; TXPK	0000C056; RXPK	0000B516; RXPK	0000B545; RXPK
No normalization	Predicted Positives	7	9	38	2	46	41	99	64	139	116	661	1182	2554
	Predicted Negatives	17	2	30	21	93	65	208	73	582	326	1093	2768	4653
With Feature Scaling	Predicted Positives	6	9	34	0	42	43	106	61	128	123	667	1173	2532
	Predicted Negatives	18	2	34	23	97	63	201	76	593	319	1087	2777	4675
With Feature Scaling + PCA	Predicted Positives	9	7	33	0	41	43	101	59	116	97	621	1144	2458
	Predicted Negatives	15	4	35	23	98	63	206	78	605	345	1133	2806	4749
With Feature Scaling + SVD	Predicted Positives	10	8	34	0	38	41	94	63	123	91	634	1136	2447
	Predicted Negatives	14	3	34	23	101	65	213	74	598	351	1120	2814	4760

Figure 4.10 - Number of predicted positives and negatives for custom version of Isolation Forest

The algorithm is configured with a training outlier rate of 0.18, which has an error rate of 99%. More details of the training parameters configuration are specified on the Appendix B.2. The results from Figure 4.6 to Figure 4.10 show that the algorithm has, in most cases, the best results while applying feature scaling and feature reduction, even though the results without normalization and feature reduction are still reasonable. These results also show that the models which are trained with a lot of data, namely 0000C056, 0000B516 and 0000B545, take much more time to be generated in this implementation of IF than in the sklearn-based version of IF, despite having better results in most devices and taking less time while applying feature scaling and feature reduction. The huge amount of time required to generate these models (over 12 minutes without FR and around 7 minutes with FR) is due to the fact that this version is adapted to Spark, and it supports distributed training in Scala using data structures inherited from the Spark library. This custom version still has always a static value for the 'contamination' and 'contaminationError' parameters, which represent the outlier rate in the training dataset that's used to compute the decision function used to classify new instances on testing, and the error margin of 'contamination', respectively. This approach can be, sometimes, slightly inaccurate, because there is no exact knowledge of the real outlier rate in the training dataset, even though it's assumed that the outlier rate is very little or zero. In the other hand, the sklearn version, 'contamination' is set to 'auto', which allows this parameter to be automatically adjusted according to the training dataset nature.

This customized version of IF has one big limitation: it couldn't be saved on MLFlow because its format was incompatible with MLFlow. It corresponds to a JVM object loaded

on Spark, which is not a Python object and can't be serialized on MLFlow, since it doesn't implement the serialization protocols required by Spark ML and Python libraries. Due to this limitation, the model was not used for real time processing, but it was still considered for comparison with the other algorithms.

4.1.4 One-Class SVM

Figure 4.11 to Figure 4.15 show the results of OCSVM, without normalization and with feature scaling, and also comparing PCA with SVD with scaling included.

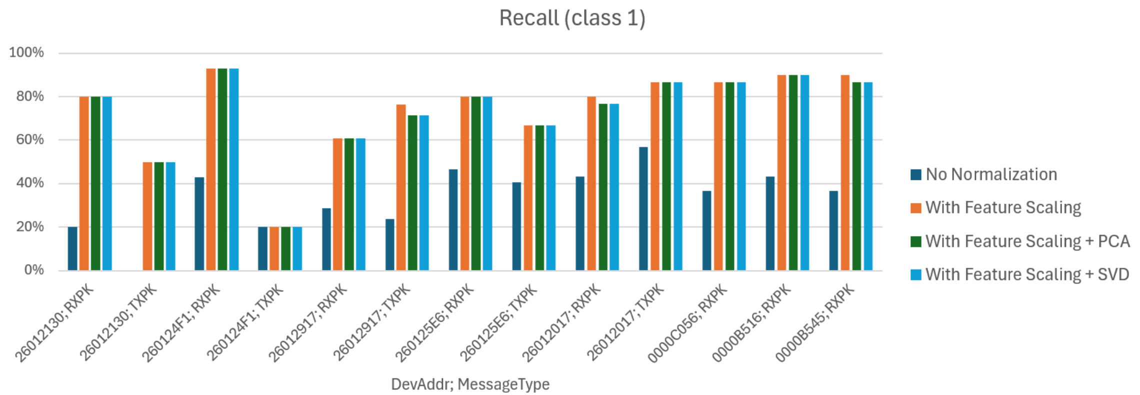

Figure 4.11 - Recall (class 1) for One-Class SVM

Figure 4.12 - Accuracy for One-Class SVM

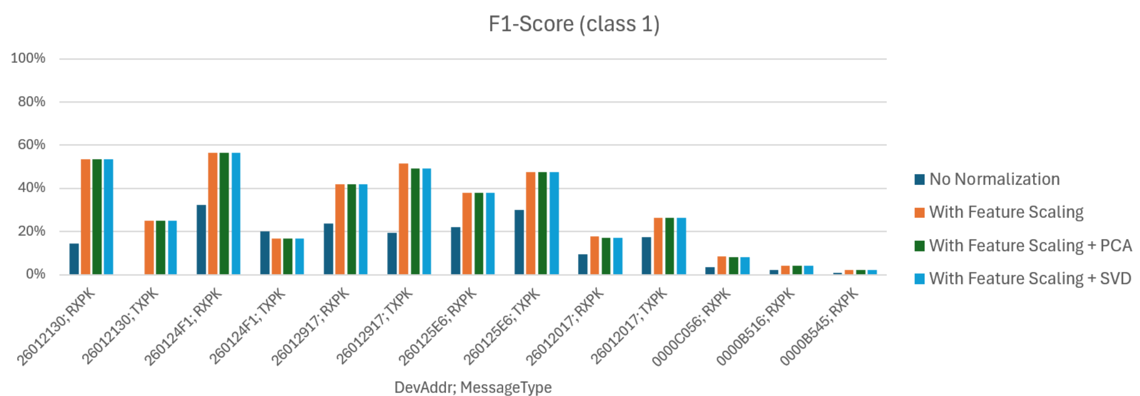

Figure 4.13 - F1-Score (class 1) for One-Class SVM

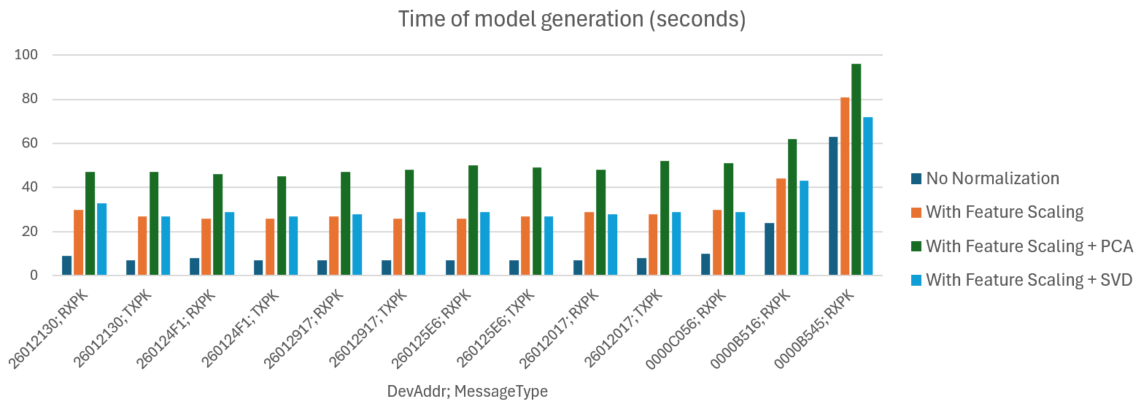

Figure 4.14 - Time of model generation for One-Class SVM

Normalization	DevAddr + Type / Values	26012130; RXPK	26012130; TXPK	260124F1; RXPK	260124F1; TXPK	26012917; RXPK	26012917; TXPK	260125E6; RXPK	260125E6; TXPK	26012017; RXPK	26012017; TXPK	0000C056; RXPK	0000B516; RXPK	0000B545; RXPK
No normalization	Predicted Positives	9	3	23	5	40	31	98	46	246	167	595	1258	2449
	Predicted Negatives	15	8	45	18	99	75	209	91	475	275	1159	2692	4758
With Feature Scaling	Predicted Positives	10	6	32	7	53	41	97	49	242	168	594	1314	2475
	Predicted Negatives	14	5	36	16	86	65	210	88	479	274	1160	2636	4732
With Feature Scaling + PCA	Predicted Positives	10	6	32	7	53	40	97	49	241	167	600	1314	2474
	Predicted Negatives	14	5	36	16	86	66	210	88	480	275	1154	2636	4733
With Feature Scaling + SVD	Predicted Positives	10	6	32	7	53	40	97	49	241	167	600	1314	2474
	Predicted Negatives	14	5	36	16	86	66	210	88	480	275	1154	2636	4733

Figure 4.15 - Number of predicted positives and negatives for One-Class SVM

The algorithm is configured with an upper bound of 1/3 for the training outlier rate. More details about the configuration of the training parameters of this algorithm are detailed on the Appendix B.1. The results from Figure 4.11 to Figure 4.15 show that the algorithm has, in most cases, the best results while applying feature scaling and feature reduction. Without normalization, OCSVM fails to detect many anomalies, and the results improve a lot when feature scaling and feature reduction are applied. Comparing PCA with SVD, they both have similar performance in terms of anomaly detection, but SVD allows a faster model generation. Without normalization, OCSVM models take the shortest time to be generated. However, in some isolated devices, the algorithm seems to have slightly better performance when applying only feature scaling, excluding feature reduction.

4.1.5 Histogram-Based Outlier Score

Figure 4.16 to Figure 4.20 show the results of HBOS, without normalization and with feature scaling, and also comparing PCA with SVD with scaling included.

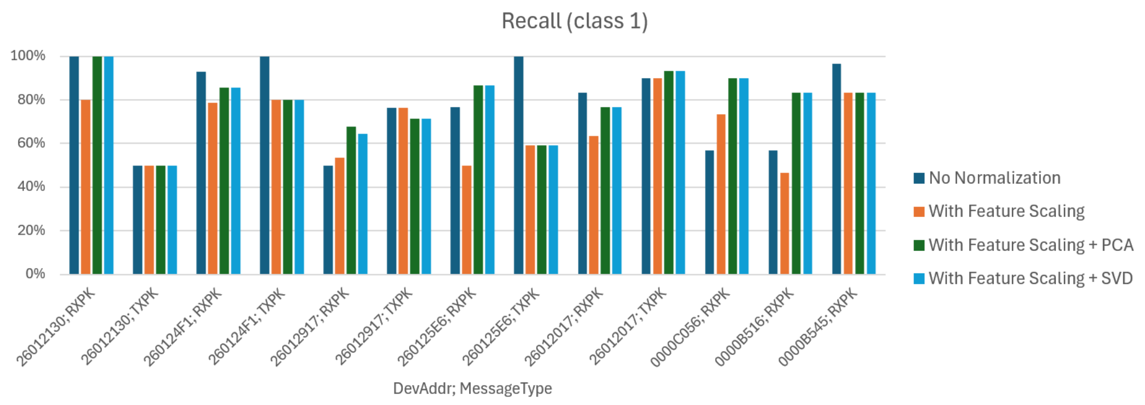

Figure 4.16 - Recall (class 1) for HBOS

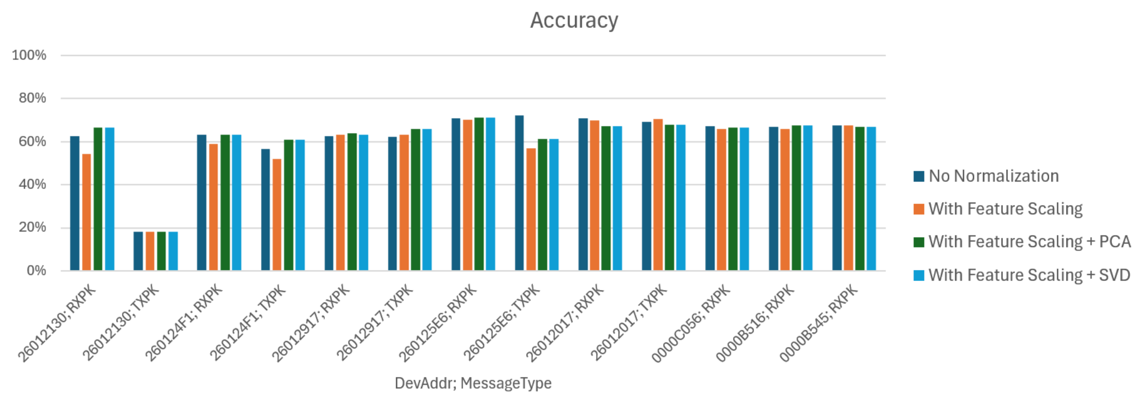

Figure 4.17 - Accuracy for HBOS

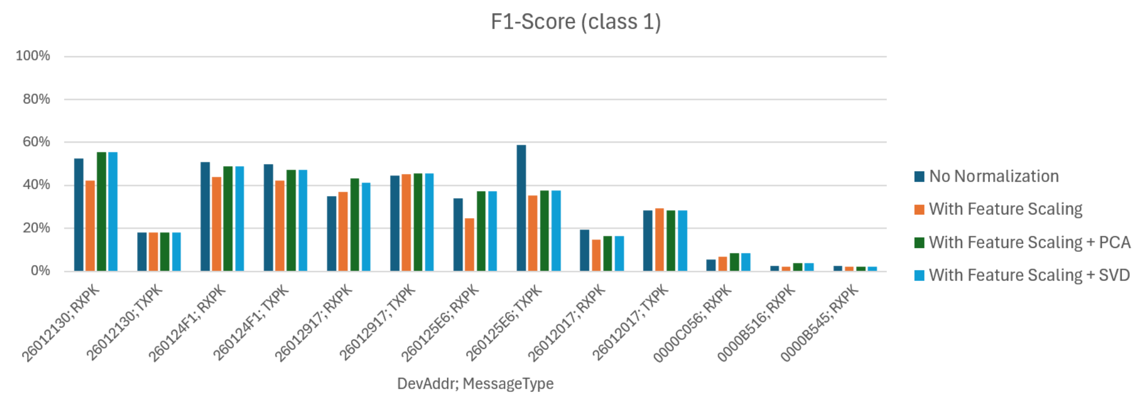

Figure 4.18 - F1-Score (class 1) for HBOS

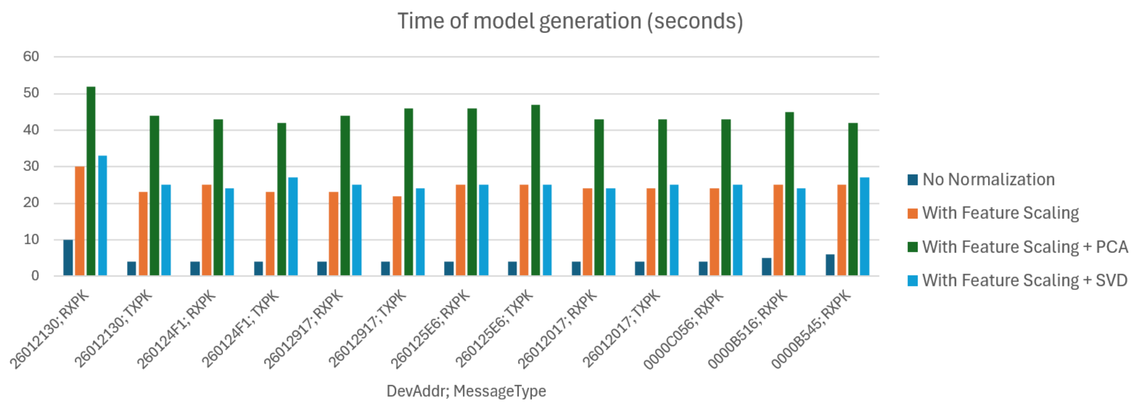

Figure 4.19 - Time of model generation for HBOS

Normalization	DevAddr + Type / Values	26012130; RXPK	26012130; TXPK	260124F1; RXPK	260124F1; TXPK	26012917; RXPK	26012917; TXPK	260125E6; RXPK	260125E6; TXPK	26012017; RXPK	26012017; TXPK	0000C056; RXPK	0000B516; RXPK	0000B545; RXPK
No normalization	Predicted Positives	14	9	37	15	52	51	105	65	230	160	576	1315	2354
	Predicted Negatives	10	2	31	8	87	55	202	72	491	282	1178	2635	4853
With Feature Scaling	Predicted Positives	14	9	36	14	53	50	91	64	225	154	612	1348	2364
	Predicted Negatives	10	2	32	9	86	56	216	73	496	288	1142	2602	4843
With Feature Scaling + PCA	Predicted Positives	13	9	35	12	64	45	110	58	253	168	611	1298	2409
	Predicted Negatives	11	2	33	11	75	61	197	79	468	274	1143	2652	4798
With Feature Scaling + SVD	Predicted Positives	13	9	35	12	59	45	110	58	253	168	611	1298	2414
	Predicted Negatives	11	2	33	11	80	61	197	79	468	274	1143	2652	4793

Figure 4.20 - Number of predicted positives and negatives for HBOS

This algorithm is configured with dynamic bin-width histograms, which means that the number of bins is set to the integer round of the square root of the number of training instances, a suggestion explained in section 2.4.2.13. It also assumes a default outlier rate of 1/3 in the training dataset. More details about the configuration of the training parameters of this algorithm are detailed on Appendix B.3. The results illustrated from Figure 4.16 to Figure 4.20 show that, as expected, HBOS stands out by its fast computation, including in large datasets. They also show that, even without normalization, HBOS has good results in many of the devices used for testing and, on this particular algorithm, the results are worse when only feature scaling is applied, and in most cases much better results are achieved without feature scaling and feature reduction, achieving recall of 100% in many devices, unlike what happened in the other tested algorithms. In fact, the results of HBOS without feature scaling and feature reduction are better than any other algorithm in any circumstance. This unexpected behaviour happens in HBOS because, as previously explained in section 2.4.2.13, HBOS treats each feature independently and assumes that the distribution of values across the entire feature space is homogeneous. In the other hand, in the input dataset pre-processing, redundant or irrelevant attributes are removed from the dataset, so that limitation of HBOS of not considering interaction between features is not a big problem. Therefore, because feature scaling scales the data to a smaller numerical range of values, it results in changes in the observed density of each bin, since rare anomalous instances are likely to move to more populated bins, which affects the results on anomaly detection, because in that case those rare anomalous instances are classified as normal

instances. When comparing PCA and SVD, SVD has better results in models trained with less data, but PCA has mostly better results on models trained with more data.

4.1.6 Local Outlier Factor

Figure 4.21 to Figure 4.25 show the results of LOF, without normalization and with feature scaling, and also comparing PCA with SVD with scaling included.

Figure 4.21 - Recall (class 1) for LOF

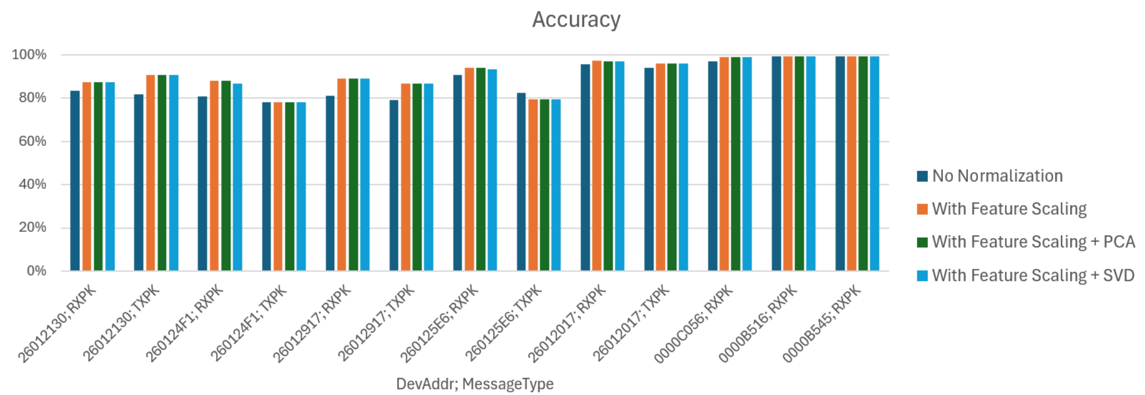

Figure 4.22 - Accuracy for LOF

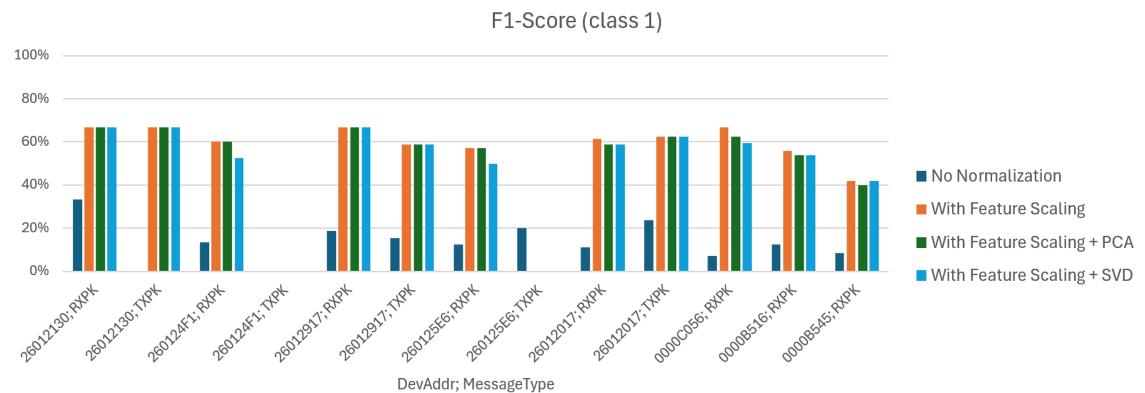

Figure 4.23 - F1-Score (class 1) for LOF

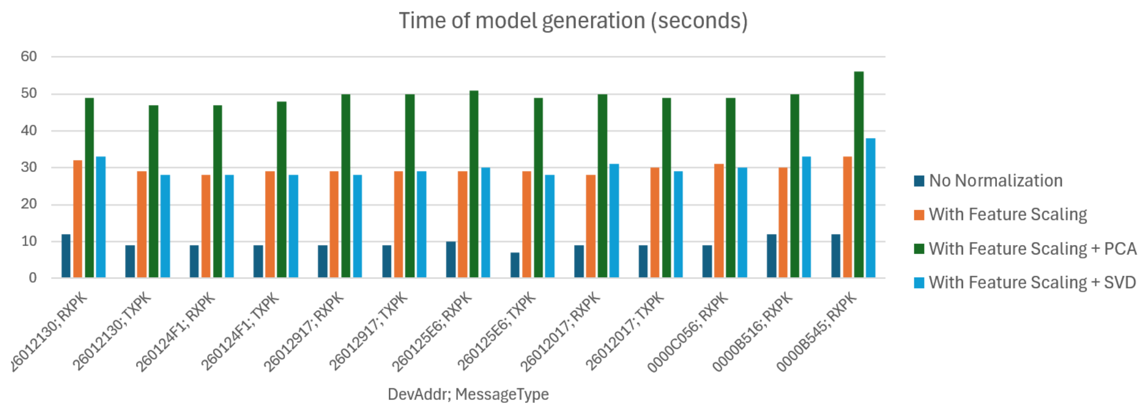

Figure 4.24 - Time of model generation for LOF

Normalization	DevAddr + Type / Values	26012130; RXPK	26012130; TXPK	260124F1; RXPK	260124F1; TXPK	26012917; RXPK	26012917; TXPK	260125E6; RXPK	260125E6; TXPK	26012017; RXPK	26012017; TXPK	0000C056; RXPK	0000B516; RXPK	0000B545; RXPK
No normalization	Predicted Positives	1	0	1	0	4	5	2	3	6	4	27	2	18
	Predicted Negatives	23	11	67	23	135	101	305	134	715	438	1727	3948	7189
With Feature Scaling	Predicted Positives	4	1	6	0	17	13	12	1	22	18	21	38	56
	Predicted Negatives	20	10	62	23	122	93	295	136	699	424	1733	3912	7151
With Feature Scaling + PCA	Predicted Positives	4	1	6	0	17	13	12	1	21	18	18	37	55
	Predicted Negatives	20	10	62	23	122	93	295	136	700	424	1736	3913	7152
With Feature Scaling + SVD	Predicted Positives	4	1	5	0	17	13	10	1	21	18	17	37	56
	Predicted Negatives	20	10	63	23	122	93	297	136	700	424	1737	3913	7151

Figure 4.25 - Number of predicted positives and negatives for LOF

This algorithm, like the sklearn-based version of IF (section 4.1.2) is configured with a dynamic training outlier rate, used to train the model, which allows the value of the corresponding parameter to be adjusted according to the nature of the training dataset. This algorithm is also set with a dynamic value for k, which is the number of the nearest neighbours of a data point, and it's set to a number between 5 and 15 which increases as the training dataset gets larger. More details about the configuration of the training parameters of this algorithm are detailed in Appendix B.4. The results from Figure 4.21 to Figure 4.25 show that in this algorithm, without normalization and feature reduction, the model fails to detect almost every introduced anomalies, and in case of devices 26012130 and 260124F1 for TXPK messages, the recall is 0%, which shows that no anomaly was detected on these cases. Device 260124F1 for TXPK achieves by far the worst results, having 0% of recall and F1-score in every pre-processing cases. When comparing PCA and SVD, PCA results in more time to generate the models, but in terms of anomaly detection, both techniques have similar performance. However, it's important to notice that, in every illustrated pre-processing cases, there are models that detect no anomalies, and particularly on device 260125E6 for TXPK messages, recall and F1-score are 0% in every cases, which shows an unexpected performance from this algorithm.

4.2 Algorithms Comparison

In summary, the results of both versions of IF assume application of Feature Scaling and SVD. The results of each algorithm assume application of Feature Scaling and SVD.

Considering these aspects, a final comparison of these algorithms was made, and the result of that comparison is illustrated from Figure 4.26 to Figure 4.30.

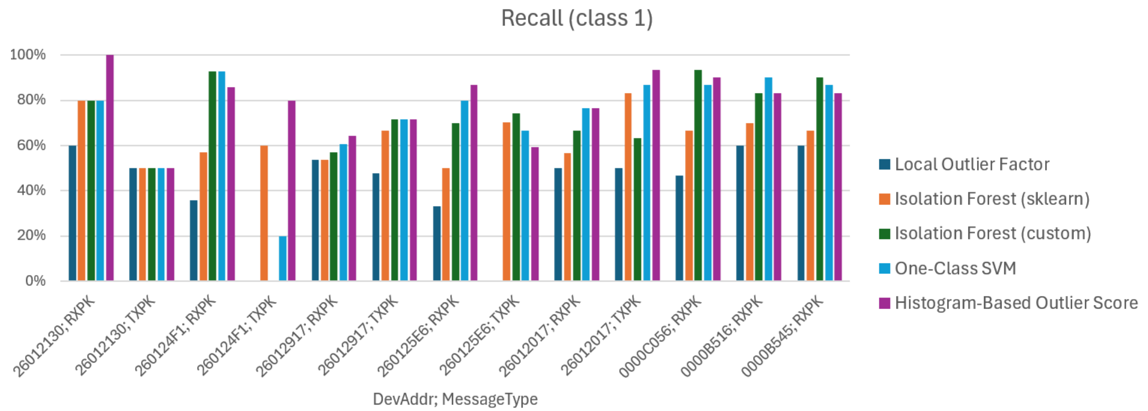

Figure 4.26 - Recall (class 1) for each algorithm

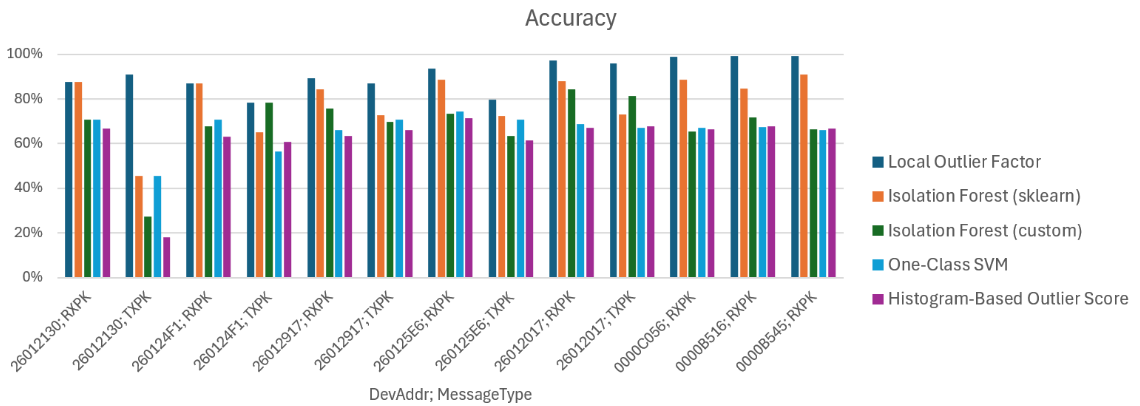

Figure 4.27 - Accuracy for each algorithm

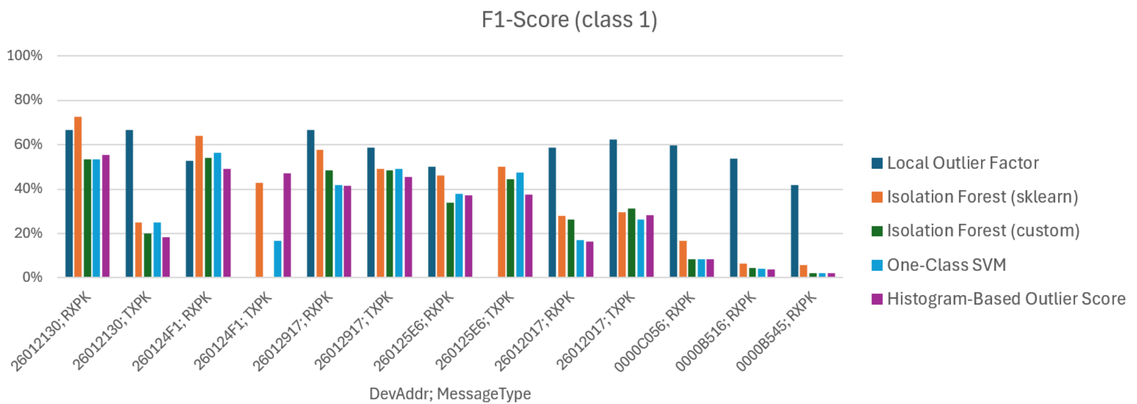

Figure 4.28 - F1-Score (class 1) for each algorithm

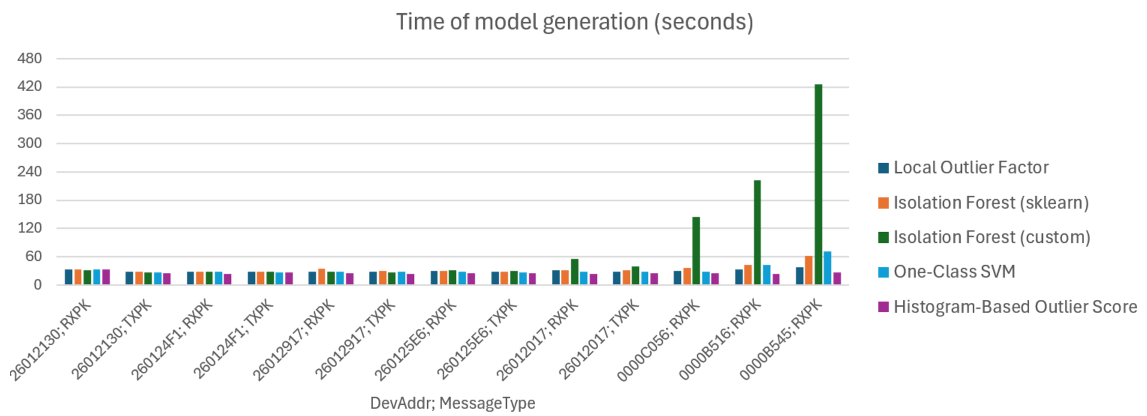

Figure 4.29 - Time of model generation for each algorithm

Algorithm	Values \ DevAddr + Type	26012130; RXPK	26012130; TXPK	260124F1; RXPK	260124F1; TXPK	26012917; RXPK	26012917; TXPK	260125E6; RXPK	260125E6; TXPK	26012017; RXPK	26012017; TXPK	0000C056; RXPK	0000B516; RXPK	0000B545; RXPK
Local Outlier Factor	Predicted Positives	4	1	5	0	17	13	10	1	21	18	17	37	56
	Predicted Negatives	20	10	63	23	122	93	297	136	700	424	1737	3913	7151
Isolation Forest (sklearn)	Predicted Positives	6	6	11	9	24	36	35	49	92	139	212	623	655
	Predicted Negatives	18	5	57	14	115	70	272	88	629	303	1542	3327	6552
Isolation Forest (custom)	Predicted Positives	10	8	34	0	38	41	94	63	123	91	634	1136	2447
	Predicted Negatives	14	3	34	23	101	65	213	74	598	351	1120	2814	4760
One-Class SVM	Predicted Positives	10	6	32	7	53	40	97	49	241	167	600	1314	2474
	Predicted Negatives	14	5	36	16	86	66	210	88	480	275	1154	2636	4733
Histogram-Based Outlier Score	Predicted Positives	13	9	35	12	59	45	110	58	253	168	611	1298	2414
	Predicted Negatives	11	2	33	11	80	61	197	79	468	274	1143	2652	4793

Figure 4.30 - Number of predicted positives and negatives for each algorithm

The results from Figure 4.26 to Figure 4.30 show that, in general, every algorithm significantly improves its results as the model is trained with more data, which is the expected behaviour in ML, because the more data the model uses for training, the better the model fits to the parameters of the environment, resulting in more robust detection of intrusions, fewer FN and a higher ability to discriminate between normal traffic and anomalous events.

HBOS is, mostly, the fastest algorithm, and HBOS and One-Class SVM are the two best algorithms. HBOS without normalization has the best results overall. In fact, the results in HBOS without normalization shown in section 4.1.5 are still better than all results illustrated from Figure 4.26 to Figure 4.30. The models trained with more data in OCSVM take more time to be generated than in the sklearn-based implementation of IF, but the custom version of IF takes too much time to generate models that use a lot of data in training (over 7 minutes in device 0000B545 for RXPK messages). LOF had the highest F1-score rates, which means that it has a low FP rate, because F1-Score is directly influenced by precision, which decreases with the increasing of FP. However, LOF achieved the worst performance because it lost more anomalies comparing to the other algorithms, which was expected because this algorithm is designed to detect anomalies only based on local densities.

4.3 Discussion

While developing the solution, the objective was to adjust several training parameters to optimize the solution to detect as much anomalies as possible, but also to avoid too much false alarms. That was done by setting the optimal value of contamination (or “nu” in some algorithms), a training parameter with a value from 0 to 1, which defines the expected outlier rate during training. This parameter directly affects the decision score, used to classify a new instance as normal or as an outlier. Considering the nature of the training dataset, which contains very little to no intrusions, contamination should be set to a very small value, such as 0.001, 0.02 or even 0.05. However, by doing so, the FN rate was very high, which means that the solution was losing many intrusions. This results in a reasonable F1-Score but low recall comparing to what is expected from an AIDS, that is to detect every anomalies coming from incoming messages. On the other hand, by setting contamination to an higher value like 0.2 or 0.33, while the solution detected much more anomalies, it had an high FP rate, resulting in too much false alarms. Sometimes, FP indicate samples that contain anomalies that were incorrectly labelled as normal samples, but in most cases, too much false alarms harm the reliability of the solution, being more difficult to realize if the alarm really indicates an intrusion or it's just mistakenly triggering an alarm related to a message that is not really anomalous. Some algorithms had the option to set the contamination to an automatic value, where the threshold used to classify an instance as normal or abnormal is dynamically computed by the algorithm during training, but while the elimination of the need of manually setting this parameter has improved the algorithm’s reliability, the FP rate was still high. This always resulted in a poor F1-Score, as the results in sections 4.1 and 4.2 illustrate. The chosen values for this parameter for each algorithm are specified in Appendix B.

It was also necessary to define an adequate number of intrusions to be manually set on the testing dataset, which is the number of samples that are affected with abnormal values for parameters. This would allow to improve the testing results, optimizing recall but also F1-Score and accuracy as possible. However, the results were never perfect, since an higher quantity of intrusions would lead to an higher F1-Score due to the increasing of TP but also a lower recall due to the increasing of FN. In the other hand, a lower quantity of intrusions would lead to an higher recall due to less FN, but also a lower F1-Score because of the lower precision, which comes from a lower TP rate giving the same amount of FP. There is a challenge on improving this solution in the future, so that there is a better balance between a good F1-Score and a perfect recall, which means to detect as much anomalies as possible, but also minimizing false alarms. This limitation is very common on previous AIDS, specially when it comes to unsupervised approaches,

and needs to be solved through an innovative approach that maybe takes advantage of the AIDS approach with other types of IDS such as specification-based IDS.

As the quantity of training data increases, the anomaly detection rate tends to improve, but the F1-Score tends to decrease. That happens because of the high value of contamination set as training parameter to optimize the recall and anomaly detection, but also because of the number of synthetic anomalies that were injected on the dataset that was very low compared to large test datasets, and that was done with the objective of maximizing the recall when testing models trained with large amounts of data. The expected results are that, as the training data keeps increasing, eventually with hundreds of thousands of training samples, the model is able to effectively classify new instances as normal or abnormal, with perfect or almost perfect precision, and the model is improved over time as it gets more data to be trained with.

Throughout the development of this implementation, another ML algorithms were initially considered. However, after a deeper study and experimental evaluation, it was concluded that these algorithms were not relevant or adequate for the problem. AE (section 2.4.2.11), for example, is primarily designed for dimensionality reduction, reconstructing the dataset in a lower-dimensional space and detecting anomalies through the reconstruction error. Although this technique can be effective in some contexts, in this case it didn’t provide the best results for anomaly detection and it wasn’t the best approach to isolate data points which possibly contained anomalies.

K-means (section 2.4.2.9) and DBSCAN (section 2.4.2.10) are two clustering algorithms. While clustering can be useful for identifying general data structures, it’s not an adequate strategy for isolating anomalous data points, as it does not provide clear insights into which clusters specifically might contain anomalies. That’s what is proved, for example, by Kaliyaperumal et al. (2024), where DBSCAN was only employed to cluster anomalies after they has already been detected by another algorithm, which is deep AE.

kNN, on the other hand, is proven to be an effective method for anomaly detection, as demonstrated by Esteves et al. (2024). However, the scikit-learn implementation of kNN proposed by these authors is a supervised algorithm, while the idea on this project is to follow an unsupervised approach, since the provided dataset does not contain labels and the objective is to detect anomalies based on network traffic patterns, and not based on existing attacks. For this reason, an unsupervised-based version of kNN was implemented and tested on PySpark for this project. Despite being functionally correct, this implementation proved to be highly inefficient in terms of computational resources, particularly when training models on large datasets. Moreover, LOF is itself an algorithm inspired by kNN, with the advantage of being an unsupervised algorithm. Consequently, to explore neighbour-based anomaly detection methods, only LOF was ultimately selected and tested in the project’s final stage.

5 Conclusions and Future Work

Throughout the development of this project, it was possible to realize that there are several implementations of anomaly detectors in IoT networks, but very few of them are specified to LoRaWAN networks, which proves that this project has a big contribution for the availability of solutions to specifically monitor LoRaWAN networks and detect any traffic deviations from each LoRaWAN message coming from any data sources.

However, the edge-computing approach, despite reducing latency, requires more computational resources, as well as the implemented and tested ML algorithms, which can be a challenge if the machine where the IDS is running does not have enough computational resources. Since a VM with 64 GB of RAM memory and 8 CPUs was used to test and execute the IDS efficiently, there is no guarantee that the same efficiency would be achieved on a machine with less computational resources than the used VM. In the implementation, Spark was configured to use 40 GB of memory because the VM had these 64 GB of RAM, and that resulted in good performance of the solution, but in a machine with, for example, 16 GB of RAM, that would not be possible, resulting in a much worse performance, being this one of the disadvantages of this solution and Apache Spark.

Furthermore, as previously mentioned, the solution's results can be improved, in order to achieve a better balance between a good F1-Score and a good recall, which means to keep detecting as much anomalies as possible, but also minimizing false alarms. Five ML algorithms were implemented and compared on LoRaWAN traffic data, with HBOS achieving, without normalization (without feature scaling and feature reduction), the best performance on testing after synthetic anomalies were injected on the test dataset. The current solution has prioritised to detect as much anomalies as possible, maximizing the recall, but that approach resulted in an high FP rate, which is exactly the current biggest problem in previous AIDS solutions. It's crucial to study, in the future, solutions similar to this that are able to minimize the FP rate while maintaining the efficacy on detecting every single anomalies coming from new data. One possible solution would be an hybrid

IDS, which combines the approaches of the SIDS, AIDS and specification-based IDS, maximizing their advantages and minimizing their limitations.

Furthermore, it's fundamental to remember that, even though this solution detects anomalies on data, it learns anomaly detection based on previous input data, and it's necessary a lot of data to make the model to detect any real anomaly from the message and also to minimize false alarms. Even with almost 37000 messages in the training dataset for the device 0000B545, despite good results, the recall was not perfect, having as best values around 80% or 90%. Before these results, the solution was tested by injecting anomalies in three parameters: SF, BW and PHYPayloadLen. In this case, some devices in some algorithms achieved perfect recall (100%), but later the implementation was changed to affect all relevant parameters with anomalous values, so that the solution would be more robust. However, after these changes, the recall in general has slightly decreased. But in the future solutions based on this solution, as the model is trained with more and more data, its effectiveness on classifying new messages as normal or abnormal will tend to improve, as expected by any ML model.

Moreover, the fact that the effectiveness of this solution depends of the input data used for training and its results are not perfect shows that, in most cases, an human review will be necessary when the solution triggers an alert indicating an anomalous message, to verify if that message really contains an anomaly.

When testing the implementation receiving messages in real time from the LoRa gateway, the results still would need to improve on other algorithms different than HBOS. When testing models were trained with few training data like 50, 100 or even 3000 samples, in OCSVM every new message was being classified as anomalies, but not in HBOS. With over 30000 training samples, the results would much likely improve, and in the end, 30000 was set as minimal number of training samples for stream processing in the project's implementation, even with HBOS, to ensure a maximized precision on message classification.

However, this project was important to assess the performance of new and innovative unsupervised ML algorithms on anomaly detection, which is an important contribution to the literature of AIDS when input datasets have no labels. Despite all the necessary improvements in the future, the main objective of the project was achieved, which was to develop ML algorithms designed to detect intrusions in messages coming from many EDs to a LoRaWAN network.

As future work, besides improving the results of this solution, there is the possibility to adapt this implementation to more IoT protocols such as NB-IoT. Moreover, this implementation could be adapted to process TXPK messages in real time coming from the TTN, by sending messages from the gateway to the TTN in order to make the TTN to send messages back to the gateway as acknowledges, and then configure the machine, where the IDS is running, as an intermediary between the gateway and the

TTN. Obviously, this solution must be constantly updated, since attackers are constantly developing new cyberattacks and strategies to disrupt services that rely on IoT technologies such as LoRaWAN.

Bibliographic References

Abdelaziz, M. T., Radwan, A., Mamdouh, H., Saad, A. S., Abuzaid, A. S., AbdElhakeem, A. A., Zakzouk, S., Moussa, K., & Darweesh, M. S. (2024). *Enhancing Network Threat Detection with Random Forest-Based NIDS and Permutation Feature Importance* (p. 36). https://doi.org/10.1007/s10922-024-09874-0

Adelantado, F., Vilajosana, X., Tuset-Peiro, P., Martinez, B., Melià-Seguí, J., & Watteyne, T. (2017). *Understanding the Limits of LoRaWAN*. https://doi.org/10.1109/MCOM.2017.1600613

Adesina, M. T., & Adesina, J. M. (2024). *Classifying Spams Using Apache Spark MLlib* (p. 19).

Alghushairy, O., Alsini, R., Soule, T., & Ma, X. (2020). *A Review of Local Outlier Factor Algorithms for Outlier Detection in Big Data Streams* (p. 24). https://doi.org/10.3390/bdcc5010001

Aouria, I., & Mezati, M. (2024). *Flink-ML: machine learning in Apache Flink* (p. 18).

Apache Kafka. (2024). *Kafka Documentation*; https://kafka.apache.org/documentation/#introduction

Apache Spark. (2024). *Apache Spark 3.5.3 - Spark Streaming Programming Guide*

Aras, E., Ramachandran, G. S., Lawrence, P., & Hughes, D. (2017). *Exploring The Security Vulnerabilities of LoRa* (p. 6). https://doi.org/10.1109/CYBConf.2017.7985777

Berahmand, K., Daneshfar, F., Salehi, E. S., Li, Y., & Xu, Y. (2024). *Autoencoders and their applications in machine learning: a survey*. https://doi.org/10.1007/s10462-023-10662-6

Breunig, M. M., Kriegel, H.-P., Ng, R. T., & Sander, J. (2000). *LOF: Identifying Density-Based Local Outliers* (p. 12). https://doi.org/10.1145/335191.335388

Butun, I., Pereira, N., & Gidlund, M. (2018). *Security Risk Analysis of LoRaWAN and Future Directions* (p. 22). https://doi.org/10.3390/fi11010003

Cao, Y., Xiang, H., Zhang, H., Zhu, Y., & Ting, K. M. (2024). *Anomaly Detection Based on Isolation Mechanisms: A Survey* (p. 10). https://doi.org/10.48550/arXiv.2403.10802

Cavallin, J., & Orpana, A. (2024). *A comparison on performance of Apache Spark and Apache Flink performing machine learning in an Apache Kafka event stream* (p. 52).

Chandra, L. S., & Singh, D. K. (2024). *An Enhanced Intrusion Detection System Model Using Machine Learning Algorithm* (p. 6). https://doi.org/10.1109/ESIC60604.2024.10481554

Chaves, A. J., Martín, C., & Díaz, M. (2023). *The orchestration of Machine Learning frameworks with data streams and GPU acceleration in Kafka-ML: A deep-learning performance comparative* (p. 20). https://doi.org/10.1111/exsy.13287

Chimphlee, W., & Chimphlee, S. (2024). *Hyperparameters optimization XGBoost for network intrusion detection using CSE-CIC-IDS 2018 dataset* (p. 10). https://doi.org/10.11591/ijai.v13.i1.pp817-826

Cuomo, F., Garlisi, D., Martino, A., & Martino, A. (2020). *Predicting LoRaWAN Behavior: How Machine Learning Can Help* (p. 18). https://doi.org/10.3390/computers9030060

Danish, S. M., Nasir, A., Qureshi, H. K., Ashfaq, A. B., Mumtaz, S., & Rodriguez, J. (2018). *Network Intrusion Detection System for Jamming Attack in LoRaWAN join procedure* (p. 6). https://doi.org/10.1109/ICC.2018.8422721

Donmez, T., & Nigussie, E. (2018). *Security of LoRaWAN v1.1 in Backward Compatibility Scenarios* (p. 8). https://doi.org/10.1016/j.procs.2018.07.143

Esteves, G. (2023). *Computação na periferia na Internet das Coisas*.

Esteves, G., Fidalgo, F., Simão, J., & Cruz, N. (2024). *Long-Range Wide Area Network Intrusion Detection at the Edge* (p. 30). https://doi.org/10.3390/iot5040040

Farrell, E. S. (2018). *Low-Power Wide Area Network (LPWAN) Overview*. https://www.rfc-editor.org/rfc/rfc8376

Fidalgo, F. (2022). *Integração de sistema de deteção de intrusões numa gateway de comunicações para a ferrovia* (p. 67). http://hdl.handle.net/10400.21/17771

Apache Flink (2024). *Flink Documentation*. https://nightlies.apache.org/flink/flink-docs-release-1.20/

García-Gil, D., Ramírez-Gallego, S., García, S., & Herrera, F. (2017). *A comparison on scalability for batch big data processing on Apache Spark and Apache Flink*. https://doi.org/10.1186/s41044-016-0020-2

García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., & Vázquez, E. (2008). *Anomaly-based network intrusion detection: Techniques, systems and challenges* (p. 11). https://doi.org/10.1016/j.cose.2008.08.003

Ghiasi, R., Khan, M. A., Sorrentino, D., Diaine, C., & Malekjafarian, A. (2024). *An unsupervised anomaly detection framework for onboard monitoring of railway track geometrical defects using one-class support vector machine* (p. 20). https://doi.org/10.1016/j.engappai.2024.108167

Gimaletdinova, A. (2024). *An In-Depth Comparative Study Of Distributed Data Processing Frameworks: Apache Spark, Apache Flink, And Hadoop Mapreduce*. https://cyberleninka.ru/article/n/an-in-depth-comparative-study-of-distributed-data-processing-frameworks-apache-spark-apache-flink-and-hadoop-mapreduce

Goldstein, M., & Dengel, A. (2012). *Histogram-based Outlier Score (HBOS): A fast Unsupervised Anomaly Detection Algorithm*.

Hasani, Z., Kon-Popovska, M., & Velinov, G. (2014). *Lambda Architecture for Real Time Big Data Analytic*. (p. 11).

Haxhibeqiri, J., De Poorter, E., Moerman, I., & Hoebeke, J. (2018). *A Survey of LoRaWAN for IoT: From Technology to Application* (p. 38). https://doi.org/10.3390/s18113995

Honfoga, A.-C., Dossou, M., & Moeyaert, V. (2024). *IoT using LoRaWAN: a security analysis* (p. 5).

Hore, S., Ghadermazi, J., Shah;, A., & Bastian, N. D. (2024). *A sequential deep learning framework for a robust and resilient network intrusion detection system* (p. 15). https://doi.org/10.1016/j.cose.2024.103928

Hou, N., Zheng, Y., & Xia, X. (2022). *CloakLoRa: A Covert Channel over LoRa PHY*. https://doi.org/10.1109/TNET.2022.3209255

Ibtissame, E., Rachida, A. A., & Abdelaziz, M. (2024). *Aquaponics Revolution: Reinforcing performance by means of Apache Spark and Apache Kafka* (p. 6). https://doi.org/10.1016/j.procs.2024.08.091

Jaiswal, J. K., & Samikannu, R. (2017). *Application of Random Forest Algorithm on Feature Subset Selection and Classification and Regression* (p. 4). https://doi.org/10.1109/WCCCT.2016.25

Jolliffe, I. T., & Cadima, J. (2016). *Principal component analysis: a review and recent developments*. https://doi.org/10.1098/rsta.2015.0202

Kaliyaperumal, P., Periyasamy, S., Periyasamy, M., & Alagarsamy, A. (2024). *Harnessing DBSCAN and auto-encoder for hyper intrusion detection in cloud computing* (p. 10). https://doi.org/10.11591/eei.v13i5.8135

Kleppman, M. (2017). *Designing Data-Intensive Applications*.

Könemann, A. (2022). *Experimental comparison between Apache Spark and Flink in heterogeneous hardware environments* (p. 71).

Liao, Y., & Vemuri, V. R. (2002). *Use of K-Nearest Neighbor classifier for intrusion detection* (p. 10). https://doi.org/10.1016/S0167-4048(02)00514-X

Liu, F. T., Ting, K. M., & Zhou, Z.-H. (2012). *Isolation-Based Anomaly Detection*. https://doi.org/10.1145/2133360.2133363

LoRa Alliance. (2015). LoRaWANTM v1.0 Specification. In *LoRaWANTM Specification* (p. 82).

LoRa Alliance. (2016a). LoRaWAN® v1.0.2 Specification. In *LoRaWAN® Specification* (p. 70).

LoRa Alliance. (2016b). LoRaWANTM v1.0.1 Specification. In *LoRaWANTM Specification* (p. 91).

LoRa Alliance. (2017). *LoRaWANTM 1.1 Specification* (p. 101).

LoRa Alliance. (2018a). LoRaWAN® v1.0.3 Specification. In *LoRaWAN® 1.0.3 Specification* (p. 72).

LoRa Alliance. (2018b). LoRaWAN® v1.0.3 Specification. In *LoRaWAN® 1.0.3 Specification* (p. 72).

LoRa Alliance. (2020). *LoRaWAN v1.0.4 Specification* (p. 90).

Lora-net. (2013). *Lora-net/packet_forwarder*. Semtech. https://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT

Mari, A.-G., Zinca, D., & Dobrota, V. (2023). *Development of a Machine-Learning Intrusion Detection System and Testing of Its Performance Using a Generative Adversarial Network* (p. 32). https://doi.org/10.3390/s23031315

Mohammed, S. H., Al-Jumaily, A., Singh, M. S. J., Jiménez, V. P. G., Jaber, A. S., Hussein, Y. S., Al-Najjar, M. M. A. K., & Al-Jumeily, D. (2024). *A Review on the Evaluation of Feature Selection Using Machine Learning for Cyber-Attack Detection in Smart Grid*. https://doi.org/10.1109/ACCESS.2024.3370911

Nayak, J., Naik, B., & Behera, H. S. (2015). *A Comprehensive Survey on Support Vector Machine in Data Mining Tasks: Applications & Challenges* (p. 18).

Noura, H., Hatoum, T., Salman, O., Yaacoub, J.-P., & Chehab, A. (2020). *LoRaWAN security survey: Issues, threats and possible mitigation techniques* (p. 37). https://doi.org/10.1016/j.iot.2020.100303

Obaido, G., Mienye, I. D., Egbelowo, O. F., Emmanuel, I. D., Ogunleye, A., Ogbuokiri, B., Mienye, P., & Aruleba, K. (2024). *Supervised machine learning in drug discovery and development: Algorithms, applications, challenges, and prospects* (p. 20). https://doi.org/10.1016/j.mlwa.2024.100576

Ozkan-Okay, M., Akin, E., Aslan, Ö., Kosunalp, S., Iliev, T., Stoyanov, I., & Beloev, I. (2024). *A Comprehensive Survey: Evaluating the Efficiency of Artificial Intelligence and Machine Learning Techniques on Cyber Security Solutions* (p. 28). https://doi.org/10.1109/ACCESS.2024.3355547

Proto, A., Miers, C. C., & Carvalho, T. C. M. B. (2024). *An Intrusion Detection Architecture Based on the Energy Consumption of Sensors Against Energy Depletion Attacks in LoRaWAN* (p. 8). https://doi.org/10.5220/0012703400003705

Qadir, J. (2023). *Cybersecurity in LoRaWAN Networks: Vulnerability Analysis and Enhancing Security Measures for IoT Connectivity* (p. 116). https://tesidottorato.depositolegale.it/handle/20.500.14242/125919

Rasamoelina, A. D., Adjailia, F., & Sincak, P. (2020). *A Review of Activation Function for Artificial Neural Network* (p. 6). https://doi.org/10.1109/SAMI48414.2020.9108717

Rashid, U., Saleem, M. F., Rasool, S., Abdullah, A., Mustafa, H., & Iqbal, A. (2024). *Anomaly Detection using Clustering (K-Means with DBSCAN) and SMO* (p. 10). https://jcbi.org/index.php/Main/article/view/598

Ruotsalainen, H., Shen, G., Zhang, J., & Fujdiak, R. (2022). *LoRaWAN Physical Layer-Based Attacks and Countermeasures, A Review* (p. 26). https://doi.org/10.3390/s22093127

Samantaray, M., Barik, R. C., & Biswal, A. K. (2024). *A comparative assessment of machine learning algorithms in the IoT-based network intrusion detection systems* (p. 16). https://doi.org/10.1016/j.dajour.2024.100478

Sarker, I. H., Abushark, Y. B., Alsolami, F., & Khan, A. I. (2020). *IntruDTree: A Machine Learning Based Cyber Security Intrusion Detection Model* (p. 15). https://doi.org/10.3390/sym12050754

Semtech (2024). *LoRaWAN Standard: Technical Document*. *LoRa® and LoRaWAN®* https://www.semtech.com/lora/lorawan-standard#technical-document

Spadaccino, P. (2023). *Challenges and Opportunities in LoRaWAN Security: Exploring Protocol Vulnerabilities, Privacy Threats and the Role of Edge Computing* (p. 127).

Stanco, G., Navarro, A., Frattini, F., Ventre, G., & Botta, A. (2024). *A comprehensive survey on the security of low power wide area networks for the Internet of Things* (p. 34). https://doi.org/10.1016/j.icte.2024.03.003

Suyal, M., & Sharma, S. (2024). *A Review on Analysis of K-Means Clustering Machine Learning Algorithm based on Unsupervised Learning* (p. 11). https://doi.org/10.33969/AIS.2024060106

Tavares, C. (2025). *IDS-software*. https://github.com/CarlosMoises20/IDS-software

Thi, T.-X. C. (2023). *Singular value decomposition and applications in data processing and artificial intelligence*. https://doi.org/10.56764/hpu2.jos.2023.2.3.34-41

Torres, N., Pinto, P., & Lopes, S. I. (2021). *Security Vulnerabilities in LPWANs - An Attack Vector Analysis for the IoT Ecosystem* (p. 28). https://doi.org/10.3390/app11073176

Valtorta, J. M., Martino, A., Cuomo, F., & Garlisi, D. (2019). *A Clustering Approach for Profiling LoRaWAN IoT Devices* (p. 16). https://doi.org/10.1007/978-3-030-34255-5_5

Verbus, J., Rosti, M., & Cozowicz, M. (2019). *Github: Linkedin - isolation-forest*. https://github.com/linkedin/isolation-forest/

Vicci, Dr. H. (2024). *The Impact of IoT on the Modern World* (p. 17). https://doi.org/10.2139/ssrn.4818308

Vigneswaran, R., Vinayakumar, Soman, & Poornachandran, P. (2018). *Evaluating Shallow and Deep Neural Networks for Network Intrusion Detection Systems in Cyber Security*. https://doi.org/10.1109/ICCCNT.2018.8494096

Vitorino, J. (2022). *Plataforma de avaliação de QoS de uma rede LoRaWAN*. http://hdl.handle.net/10400.21/16197

Yang, X., Karampatzakis, E., Doerr, C., & Kuipers, F. (2018). *Security Vulnerabilities in LoRaWAN*. https://doi.org/10.1109/IoTDI.2018.00022

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). *Resilient Distributed Datasets - A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Zaheer, R., Hanif, M. K., Sarwar, M. U., & Talib, R. (2025). *Evaluating the Effectiveness of Dimensionality Reduction on Machine Learning Algorithms in Time Series Forecasting*. https://doi.org/10.1109/ACCESS.2025.3551741

Zarpelão, B. B., Miani, R. S., Kawakani, C. T., & de Alvarenga, S. C. (2017). *A survey of intrusion detection in Internet of Things* (p. 13). https://doi.org/10.1016/j.jnca.2017.02.009

Appendices

A LoRa Gateway Configuration

The configured LoRa gateway was a provided MikroTik LoRa gateway, illustrated on Figure A.1. This LoRa gateway was already previously configured to periodically receive LoRaWAN messages. To configure the LoRa gateway to forward the received LoRaWAN messages to a chosen destination, the first step was to find a port that was active and working. After that, the LoRa gateway was connected to that port using an Ethernet cable (the yellow cable visible on Figure A.1), and Internet connection was provided to the gateway using that port to allow the gateway to be configured to forward its LoRaWAN messages to a chosen destination.

Figure A.1 - MikroTik LoRa Gateway

To configure the gateway, WinBox was installed on a machine. WinBox is the official graphical configuration utility for MikroTik RouterOS devices. It provides an intuitive user interface to manage and configure routers, including the LoRa gateway used in this project. Through WinBox, the gateway was configured to forward UDP traffic to the VM where IDS was running. The software was downloaded from the official MikroTik website. To connect to the gateway using WinBox, it was necessary to use its IP address, username and password. To find the gateway IP address, the gateway was connected to the computer using the Ethernet cable and then, on the command line of the machine

where WinBox is installed, it was executed the command that displays a detailed overview of the network configuration for all network adapters on the system, that on Windows is ipconfig /all, and then the information corresponding to Ethernet connection to the gateway was identified, which includes the gateway IP address.

After connecting to the gateway on WinBox, the next step was to configure the destination of traffic coming from the LoRa gateway. To do so, in the WinBox interface, on IoT -> LoRa -> Servers, the server which corresponds to the LoRaWAN traffic destination was configured, with a random name, the address corresponding to the DNS name or IP address of the VM where IDS is running, the Up Port, from where traffic coming from the gateway is forwarded, and Down Port, from where the traffic coming to the gateway is forwarded. The three protocols available as options to use for the server were UDP, LNS and CUPS, and UDP was chosen. An example of server configuration is illustrated on Figure A.2.

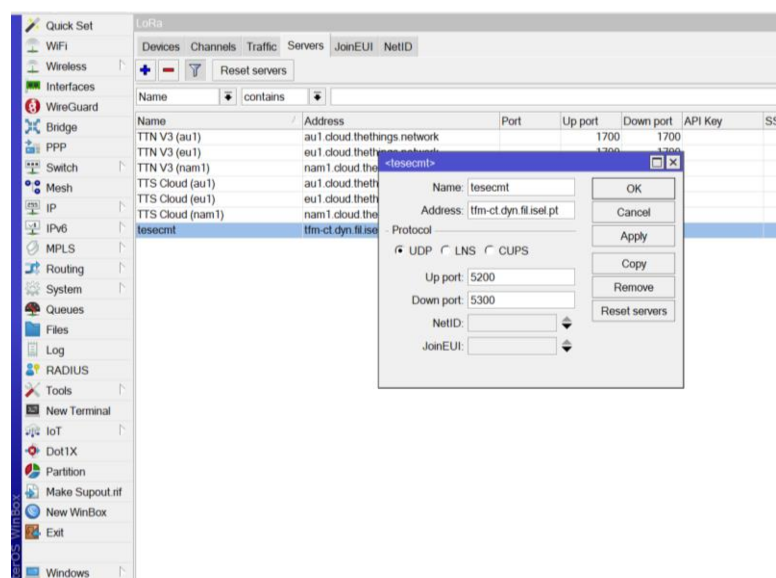

Figure A.2 - Server configuration for LoRaWAN message destination ("tesecmt")

After configuring the server, the gateway was configured to forward its messages to the configured server. To do so, on IoT -> LoRa -> Devices, a new device was added, with a random name, with the Gateway ID and the Firmware ID that are visible on the gateway. Channel Plan was set to EU 868, Band was set from 863 to 870 and the NS was configured to be the one configured in the previous step. An example of gateway configuration is illustrated on Figure A.3.

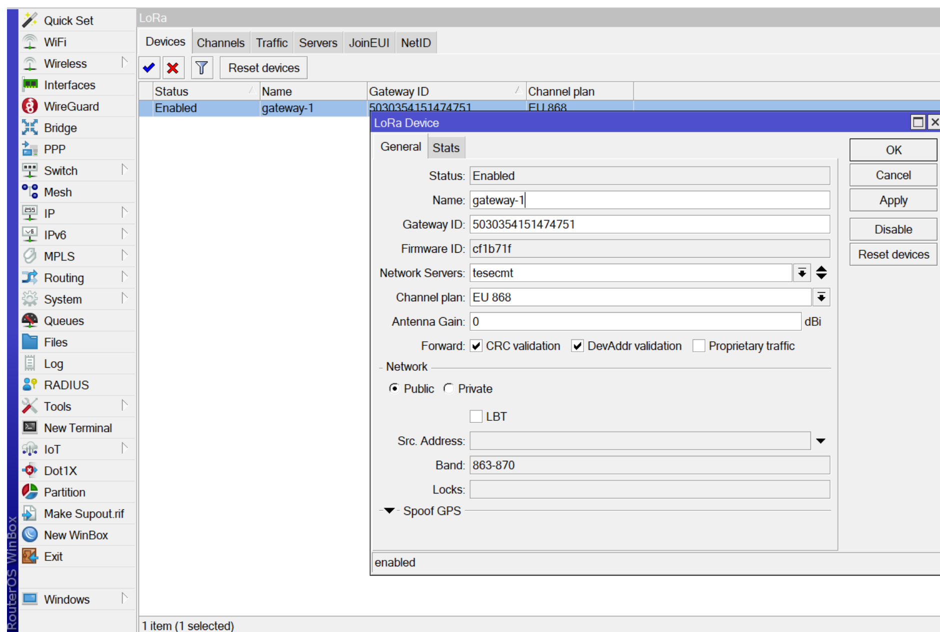

Figure A.3 - Device configuration for LoRa gateway ("gateway-1")

The UDP socket was bound to IP address 0.0.0.0, to allow listening from all network interfaces, and to port 5200, which corresponds to the Up Port configured on the LoRa gateway (Figure A.2), the port through which LoRaWAN messages are forwarded.

B Training Parameters of ML Algorithms

B.1 One-Class Support Vector Machine

The implementation of the One-Class SVM training model contains the following training parameters, according to its SkLearn specification:

- **Tol:** a float value which represents the tolerance for stopping criterion; it decides how small the gradient in the optimization algorithm has to be to consider that it converged; a smaller 'tol' will result in a more accurate training, but also slower; therefore, it was manually set to 10^{-12} to ensure more accuracy without slowing down the training too much;
- **Nu:** a float value which represents the upper bound of the contamination rate in the dataset, and also the lower bound of the fraction of support vectors; even though we consider that the dataset contains little to no anomalies, we are not sure about the anomaly rate in the training dataset and, because this parameter corresponds to the upper bound of the anomaly rate and not the exact anomaly rate in the training dataset, it was manually set to 1/3, so that it would allow to minimize FN efficiently in this algorithm, detecting more anomalies in the dataset;
- **Kernel:** string value which represents the kernel type to be used in the algorithm. For this parameter, the default value was used, which is ‘rbf’. The ‘rbf’ kernel function is the only one which allows to learn non-linear relationships and detect rare outliers;
- **Degree:** string value which represents the degree of the polynomial kernel function (kernel=’poly’), therefore it’s not used by the model;
- **Gamma:** float value which represents the kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’, and it was set to its default value (0.0) since there was no found benefit on changing it to a specific value;
- **Coef0:** string value; this parameter is not used since it’s only significant in ‘poly’ and ‘sigmoid’ kernel functions. Therefore, it’s set to its default value (0.0);

- **Shrinking:** boolean that defines is the shrinking heuristic is used (True) or not (False). It was set to its default value (True);
- **Cache_size:** integer value which represents the size of the kernel cache in MB. It was set to its default value (200) since there was no found benefit on changing it to a specific value;
- **Verbose:** this boolean enables verbose output (True when enabled, False otherwise). This takes advantage of a per-process runtime setting, and if enabled, may not work properly in a multithread context. Hence, this was set to its default value (False);
- **Max_iter:** integer value which represents the hard limit on iterations within solver; it was set, by default, with -1 to indicate that this limit isn't applied.

B.2 Isolation Forest

The implementation of the sklearn-based IF training model contains the following training parameters:

- **n_estimators:** the number of base estimators in the ensemble, which can also be considered the number of trees used on training. This integer value was set according to the equation (B.1), where T is the number of trees and N is the number of training samples used to create the model. This would allow the model not to lose effectiveness when being trained with more data but also not to slow down its performance too much;

$$T = \min\left(100 + int\left(\frac{N}{5}\right), 7000\right) \quad \text{(B.1)}$$

- **n_jobs:** integer value which represents the number of jobs running in parallel while executing the model training. It's set to -1 to indicate that this number corresponds to the number of the processors of the machine, which speeds up the processing;
- **random_state:** the seed, used for data reproducibility and to control the pseudo-randomness of the selection of the feature and split values for each branching step and each tree in the forest. It ensures that, in multiple function calls, the feature selection results are the same, which is very important to compare results using different values for input model training parameters. The value is set to 42 but could be another integer value;
- **max_samples:** integer value which represents the number of samples to draw to train each base estimator; it's set to "auto" to be adapted according to the number of training samples, following the equation (B.2), where N is the number of training samples;

$$\max_samples = \min(256,\ N) \quad \text{(B.2)}$$

- **contamination:** the outlier / anomaly rate in the training dataset. This could be a float between 0 and 1 or a string, but since there isn't a exact idea of the outlier rate in the training dataset, this parameter was set to 'auto', being this parameter determined according to what was stated by Liu et al. (2012);
- **max_features:** the number of features to draw to train each base estimator. It's a float between 0 and 1 and it was set, by default, to 1.0, since feature subsampling (value less than 1.0) would lead to a longer runtime and it's not adequate for this case;
- **bootstrap:** boolean that determines if the individual trees are fit on random subsets of the training data sampled with replacement (True), or if sampling without replacement is performed (False). This parameter is set, by default, to False, since replacement on sampling isn't necessary and would require more computational resources, and also because IF isolates points in random subsets, and diversity between samples is more helpful than repeating them. This avoids overfitting and generates more independent trees;
- **verbose:** integer value which controls the verbosity of the tree building process, and it's set, by default, to 0; its meaning is similar to verbose on OCSVM (section B.1);
- **warm_start:** when this boolean is set to True, the solution of the previous call is used to fit and add more estimators to the ensemble; in this case, this is not the objective, the objective is to create a new model when the fit function is called, so this parameter is set, by default, to False.

The custom version also contains some of these parameters according to the implementation in the GutHub repository published by Verbus et al. (2019), and these are num_estimators, max_samples, contamination, max_features, bootstrap and random_seed. However, contamination on this version is not set to 'auto', it can only be defined as a static value, therefore the value 0.18 was chosen for this parameter. Additionally, this version includes the following parameters:

- **contaminationError:** the error allowed when calculating the threshold required to achieve the specified contamination fraction. The real contamination, which is the anomaly rate on the training dataset, must rely between (contamination – (contamination * contaminationError)) and (contamination + (contamination * contaminationError)). contaminationError is set to 0.99, which means that the real contamination must be on the range [0.18 – 0.18*0,99, 0.18 + 0.18*0,99], which is [0,0018, 0,3582]. This gives the model a large error margin while computing the specified contamination rate, ensuring that the contamination doesn't hit too large or too small values;

- **featuresCol:** the name of the features column, as string, which corresponds to the feature vector and is set to “features”. This column is used for training and scoring;
- **predictionCol:** the name of the column, as string, which represents the predicted label, which is set to “prediction”. This column is appended to the input dataframe upon scoring;
- **scoreCol:** the name of the column which represents the outlier score. This column is not used for evaluation purposes, but it’s appended to the input dataframe upon scoring;
- **n_estimators:** the number of trees in this customized implementation was defined differently, according to the equation (B.3), because this implementation is more resource-consuming and more trees would degrade its performance.

$$T = \min\left(100 + int\left(\frac{N}{3}\right), 3000\right) \quad \text{(B.3)}$$

B.3 Histogram-Based Outlier Score

The Pyod-based implementation of HBOS contains the following training parameters:

- **n_bins:** the number of used bins for each feature. It can be an integer value or a string, and it was manually set to the integer round of the square root of the number of training samples. It could also be set to “auto” to estimate an optimal number of bins for each feature, but this would cause, sometimes, cause issues related to the algorithm implementation. Therefore, this dynamic value was used, being adequate for every features in general;
- **alpha:** this regularizer prevents overflow. It’s a float between 0 and 1 and it was set to 0.1 by default since it was not necessary to define a different value to improve the model results;
- **contamination:** similar to contamination in section B.2; it was set to 1/3 to optimize the model efficacy on detecting anomalies, maximizing the recall and minimizing FN. There is no knowledge of the exact outlier rate in the training dataset and it can be only defined as a static value, and this is a value that allows to detect as much anomalies as possible, directly affecting the decision score, even though there are little to no anomalies on the training dataset;
- **tol:** a parameter which decides the flexibility while dealing with the samples falling outside the bins. It’s a float between 0 and 1 and it was set to 10^{-20} to ensure more accuracy on training without slowing down the performance too much;

B.4 Local Outlier Factor

The implementation of the sklearn-based LOF training model contains the following training parameters:

- **n_neighbors:** the number of neighbours to use, which is the number of the closest points to a new instance that is used to determine if that instance is a normal data point or an outlier. This integer value is a value between 5 and 15, defined according to the equation (B.4), where N is the number of training samples. This allows the model to adapt to the increasing of the training data without losing efficacy in detecting outliers;

$$n_neighbors = \max(5, \min(round(N * 0.01),\ 15) \quad \text{(B.4)}$$

- **n_jobs:** equivalent to n_jobs on section B.2;
- **novelty:** boolean that indicates if only unknown anomalies are detected (True) or the anomaly detection logic includes detecting known anomalies; this is an interesting approach that was tested to evaluate the model results, so this parameter was set to True;
- **contamination:** equivalent to contamination in section B.2;
- **p:** integer value which defines the metric to be used to calculate the distance between the new data instance and its closest neighbours; it was set to 2, by default, to use the standard Euclidean distance, similar to the distance metric used in kNN (section 2.4.2.1);
- **metric:** string value corresponding to the metric to use for distance computation; this was considered because its default value was “minkowski”, used when p=2;
- **leaf_size:** integer value corresponding to the leaf, which is size passed to BallTree or KDTree. This can affect the speed of the construction and query, and also the memory required to store the tree. Therefore, it was set, by default, to 30, to avoid too much computational cost;
- **algorithm:** string value corresponding to the algorithm used to compute the nearest neighbours; it was, by default, set to “auto” to define the most adequate algorithm according to the nature of the training dataset;
- **metric_params:** dictionary with additional keyword arguments for the metric function. This was not specified for this case, since there was no found benefit by defining it.

B.5 Dataset Structure

This section contains the structure of each type of LoRaWAN messages in the provided datasets, including the details of each attribute of each type of LoRaWAN message.

Section B.5.1 covers the structure of RXPK messages and section B.5.2 covers the structure of TXPK messages.

B.5.1 RXPK Messages

In the provided dataset, each line is a JSON object which contains the following parameters, which are mostly specified by Lora-net (2013) or LoRa Alliance (2018), and these were also previously explained in Chapter 2, namely LoRaWAN attributes:

- **fromip:** IP address and port from which messages were sent (in format IP:port, for example 191.22.22.22:2200 where 191.22.22.22 is the IP address and 2200 is the port);
- **ip:** IP address from which the messages were sent;
- **port:** port from which the messages were sent;
- **recv_date:** date and time when the messages were sent, in "YYYY-MM-DD HH:MM:SS" format (for example, "2020-06-03 23:00:20");
- **totalrxpk:** the total number of RXPK messages sent;
- **type:** type of message, in string, which in this case is always "rxpk";
- **rxpk:** array which contains one or more RXPK messages, where each one of them contain the following parameters:
    - **AppEUI:** 8-byte global identifier of the application. It's only present on JR messages and it uniquely identifies the entity able to process the JR. In the datasets, this field is present in hexadecimal;
    - **AppNonce:** 3-byte random value or some fort of unique ID provided by the NS and used by the ED to derive the two session keys NwkSKey and AppSKey. It's only present on JA messages. In the datasets, this field is present in hexadecimal;
    - **CFList:** optional list of channel frequencies that only come in JA messages, which is region specific. It contains up to 5 channel frequencies, since it has up to 15 bytes and each channel frequency has, generally, 3 bytes. In the datasets, this field is present in hexadecimal;
    - **DLSettings:** 1-byte field that is present only on JA message and contains the downlink configuration. In the datasets, this field is present in hexadecimal;
        - **DLSettingsRX1DRoffset:** field that corresponds to the 2nd to 4th bit of DLSettings, counting from the left. It sets the offset between the uplink data rate and the downlink data rate used to communicate with the ED on the first reception slot. By default, this offset is 0. In the datasets, this field is present in decimal;

    - **DLSettingsRX2DataRate:** field that corresponds to the last byte of DLSettings, counting from the left. It corresponds to the data rate in the second reception slot. In the datasets, this field is present in decimal;
- **DevAddr:** 4-byte field corresponding to the ED address that sends the message. This address is attributed by the NS to that ED when it joins the NS successfully, and it identifies the ED within the corresponding network. Hence, it's present in every LoRaWAN messages except JR messages. In the datasets, this field is present in hexadecimal;
- **DevEUI:** globally unique identifier of the ED. In the datasets, this field is present in hexadecimal;
- **DevNonce:** random value that is only present on JR messages. It's a counter For each ED, the NS keeps track of a certain number of DevNonce values used by the ED in the past, and ignores JR with any of these DevNonce values from that ED, in order to prevent replay attacks. In the datasets, this field is present in hexadecimal;
- **Direction:** the direction from which the message is sent, as string. As a rule, RXPK messages' direction is "up";
- **FCnt:** 2-byte field corresponding to the frame counter of the ED, which corresponds to the number of data frames sent between the ED and the NS and is a 16-bits value that is used directly as the counter value, possibly extended by leading zero octets if required. In the datasets, this field is present in hexadecimal;
- **FCtrl:** 1-byte field corresponding to the frame control of the frame header; this field contains parameters that are used to control specific behaviours of the frame, such as ADR and ACK. In the datasets, this field is present in hexadecimal
    - **FCtrlADR:** represents the first bit in FCtrl counting from the left, which corresponds to the frame control adaptive data rate and is False if the bit is set to 0, and True if it's set to 1. If this bit is set to 1, the network will control the data rate of the ED through the appropriate MAC commands. In the datasets, this field is present in boolean;
    - **FCtrlACK:** the third bit in FCtrl counting from the left. It's False if the bit is set to 0, and True if it's set to 1. If this bit is set to 1, when some of the parts receive a confirmed data message, the receiver shall respond with a data frame. In the datasets, this field is present in boolean;

- **FHDR:** frame header, present on messages that are neither JR nor JA; it's part of MAC Payload and it contains DevAddr, FCtrl, FCnt and FOpts. In the datasets, this field is present in hexadecimal;
- **FOpts:** field with up to 15 bytes which contains frame options used to transport MAC commands; its size corresponds to the value of FOptsLen, which is the last byte of FCtrl counting from the left. In the datasets, this field is present in hexadecimal;
- **FPort:** optional 1-byte port field, but it must be present if the frame payload is not empty; if MAC commands are present in FOpts, port 0 cannot be used. Otherwise, it can be used to indicate that the frame payload only contains MAC commands; values 1 to 223 are application-specific, value 224 is dedicated to LoRaWAN MAC layer test protocol, and values 225 to 255 are reserved for future standardized application extensions. In the datasets, this field is present in hexadecimal;
- **FRMPayload:** MAC frame payload, which is optional and can have up to N bytes and is encrypted with NwkSKey or AppSKey according to FPort. In the datasets, this field is present in hexadecimal;
- **MACPayload:** the complete MAC Payload, which has from 1 to M bytes and contains FHDR and optionally FPort and FRMPayload. In the datasets, this field is present in hexadecimal;
- **MHDR:** 1-byte field, corresponding to the MAC header field, which contains the message type (first 3 bits counting from the left) . In the datasets, this field is present in hexadecimal;
- **MIC:** 4-byte field that corresponds to the message integrity code, used to check the integrity of the message. It's calculated over MHDR, FHDR, FPort and FRMPayload using the NwkSKey and AES-128 cipher mode. In the datasets, this field is present in hexadecimal;
- **MessageType:** 3-bit field inside MHDR which corresponds to the type of LoRaWAN message. It can be Join Request (000), Join Accept (001), Unconfirmed Data Up (010), Unconfirmed Data Down (011), Confirmed Data Up (100), Confirmed Data Down (101), RFU (110) and Proprietary (111) . In the datasets, this field is present in hexadecimal;
- **NetID:** the unique network identifier, which is only present on JA messages. In the datasets, this field is present in hexadecimal;
- **PHYPayload:** the entire payload of the message, which contains MHDR, MAC Payload and MIC. In the datasets, this field is present in hexadecimal;

- **RxDelay:** 1-byte field corresponding to the delay between TX and RX, only present on JA messages. In the datasets, this field is present in hexadecimal;
    - **RxDelayDel:** the last byte of RxDelay counting from left, in decimal;
- **Aesk:** parameter not found in the specifications;
- **Brd:** parameter not found in the specifications;
- **Chan:** “IF” concentrator channel used for RX, it’s an unsigned integer;
- **Codr:** LoRa ECC coding rate identifier, present as a fraction in a string;
- **Data:** it’s equivalent to PHYPayload, but encoded with Base64. When the gateway sends messages, it uses this parameter for more security in information transport;
- **Datr:** data rate, it contains the SF and BW in a string with a specific format. For example “SF7BW125” indicates SF=7 and BW=125.
- **Freq:** the frequency used for message transmission;
- **Jver:** parameter not found in the specifications;
- **Lsnr:** LoRa Signal Noise Ratio;
- **Modu:** modulation used; it can be “LoRa” or “FSK”, but in this case it’s always “LoRa”
- **Rfch:** “RF” channel concentrator used for RX, it’s an unsigned integer;
- **Rsig:** array of elements with several parameters inside it
    - **Ant:** parameter not found in the specifications, but this field represents the antenna;
    - **Chan:** equal to “Chan” outside “rsig”. It’s an optional alternate channel;
    - **Dbg1:** parameter not found in the specifications;
    - **Dbg2:** parameter not found in the specifications;
    - **Etime:** parameter not found in the specifications;
    - **Foff:** parameter not found in the specifications;
    - **FtDelta:** parameter not found in the specifications, but this field indicates how long the packet is in the network;
    - **Ftime:** parameter not found in the specifications;
    - **Ftstat:** parameter not found in the specifications;
    - **Ftver:** parameter not found in the specifications;
    - **Lsnr:** LoRa Signal Noise Ratio;
    - **Rssic:** parameter not found in the specifications;
    - **Rssis:** parameter not found in the specifications;
    - **Rssisd:** parameter not found in the specifications;
- **Rssi:** received signal strength indicator;

  - **Size:** RF packet payload size in bytes, as an unsigned integer;
  - **Stat:** CRC status, which is 1 if it's all fine, -1 if a fail occurs and 0 if there is no CRC;
  - **Time:** date and time when the messages were sent, in timestamp format (for example, "2020-06-15T23:02:09.058336Z");
  - **Tmms:** GPS time of packet RX;
  - **Tmst:** internal timestamp of "RX finished" event, as a 32-bit unsigned integer.

B.5.2 TXPK Messages

In the provided dataset, each JSON object corresponding to TXPK messages contains the following attributes, mostly specified by Lora-net (2013) or LoRa Alliance (2018):

- **AppNonce, CFList, DLSettings, DLSettingsRX1DRoffset, DLSettingsRX2DataRate, DevAddr, Direction (always "down" in TXPK messages), FCnt, FCtrl, FCtrlACK, FCtrlADR, FHDR, FOpts, FPort, FRMPayload, MACPayload, MHDR, MIC, MessageType, NetID, PHYPayload, RxDelay, RxDelayDel, fromip, ip, port, recv_date, type (always "txpk" in TXPK messages):** all these parameters are also in RXPK messages, as detailed in section B.5.1;
- **FreqCh4, FreqCh5, FreqCh6, FreqCh7 and FreqCh8:** each one of the channel frequencies present on CFList, if CFList is specified;
- **Txpk:** struct element with the following parameters inside:
  - **Codr, data, datr, freq, modu, rfch, size, tmst:** all these parameters are also in RXPK messages, as detailed in section B.5.1;
  - **Imme:** Boolean that indicates if packet is immediately sent (True) or not (False). If packet is immediately sent, tmst and time are ignored;
  - **Ipol:** Boolean that indicated if there is LoRa modulation polarization inversion (True) or not (False);
  - **Ncrc:** Optional boolean that indicated if the CRC of the physical layer is disabled (True) or not (False);
  - **Powe:** TX output power in dBm, as an unsigned integer.