SAPIENZA
UNIVERSITÀ DI ROMA

Privacy monitoring of LoRaWAN devices through traffic stream analysis

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Cybersecurity

Candidate

Francesco Terenzi
ID number 1702217

Thesis Advisor

Prof. Francesca Cuomo

Co-Advisor

Ing. Pietro Spadaccino

Academic Year 2020/2021

Thesis not yet defended

Privacy monitoring of LoRaWAN devices through traffic stream analysis
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: January 3, 2022

Author's email: info@francescoterenzi.it

Dedicata alla mia famiglia.

Abstract

LoRaWAN is a wireless technology developed to transmit over long distances using low power. It runs over the proprietary LoRa radio modulation and provides fundamental IoT requirements such as bi-directional communication, end-to-end security, mobility, and localization services. Despite LoRaWAN guarantees confidentiality and integrity of transmitted data, the wireless nature of the medium causes that an eavesdropper, listening to the network communications, can collect unencrypted elements stored in the packets. In particular, it can obtain two sensible metadata elements, called DevAddress and DevEUI. Since the association between these elements can involve privacy issues, LoRaWAN forces endpoints to expose their DevEUI only during the association procedure. In the first part of this work, we prove how an adversary, exploiting well-known vulnerabilities of LoRaWAN, can link them anyway. Then we explain the consequences for the privacy of devices and users that joined the network and propose PIVOT (Privacy-Monitoring), an analyzer system for LoRaWAN that detects in real-time vulnerable endpoints. Furthermore, we explain how the metrics used in PIVOT can support the operator in applying adequate countermeasures. Finally, we test our scheme on a simulated LoRaWAN application and examine the results obtained.

Contents

Chapter 1

Introduction

From asset tracking and smart metering to air quality monitoring, industries make use of sensors to collect large amounts of data to improve efficiency and lower costs. Businesses face complex and remote environments where IoT applications require long-range connectivity, low data rate, and low energy consumption. Traditional technologies don't meet the coverage and power requirements needed [19]. For example, Bluetooth is intended only for short-range uses and Wi-Fi is too expensive, power-hungry, and encounters significant coverage gaps. Furthermore, mesh networks, because of their excessively complex network, don't scale beyond medium-range applications.

Created for IoT applications that transmit small amounts of data, Low-Power Wide Area Network (LPWAN) is a class of wireless technologies with a range that varies from a few kilometers in urban areas to over 15 km in rural settings [8]. LPWAN lightweight protocols reduce the complexity and costs of sensors, which, consequently, can operate for more than ten years, even if equipped with small batteries [27].

One prominent LPWAN modulation technique is LoRa. Developed by Cycleo of Grenoble and acquired by Semtech, LoRa works on the physical layer of the OSI model and enables a link for long-range communications. LoRaWAN is one of the various protocols created to define the upper layer of LoRa. It delineates the system architecture of the network and manages the communication frequencies, data rate, and power for all devices. Currently, 163 network operators in 177 countries [10] adopt LoRaWAN, and its coverage is significantly expanding.

1.1 Weaknesses in LoRaWAN

The wide communication range of LoRaWAN causes the messages sent in the network can be intercepted by unauthorized eavesdroppers, even hundreds of meters away from endpoints. Despite LoRaWAN having built-in security mechanisms to protect the confidentiality of user data [11], such as encrypting the payload of messages exchanged, some relevant parameters remain exposed. Indeed, since the header of the packets is left-in-clear, potentially lifting metadata is shown, such as two LoRaWAN identifiers, the DevAddress and the DevEUI. The DevAddress is a temporary address generated by the network where devices are operating, while the DevEUI is a global, unique identifier from the manufacturer. Since the DevEUI represents a source of information about endpoints and their constructors, LoRaWAN's devices reveal it the least possible, preventing its association with other fields and, in particular, with the DevAddress. Indeed, endpoints expose their DevEUI only during the association phase, called OOTA, and show their current DevAddress only in uplink messages they send to the Network Server. Since cannot be a packet on the network with both of them exposed in the header, the connection between these two identifiers, known only by the network, is theoretically unlikely to externals. Nevertheless, several studies discovered several methodologies an attacker can exploit to connect these two elements. This weakness compromises the users' privacy and companies of the targeted network: by linking the DevEUI to the DevAddr, the intruder obtains further knowledge about the devices and their activities. It can discover associated applications, the manufacturer that produced them, the type of equipment, and their purposes.

In this work, we present PIVOT (Privacy-Monitoring), a privacy-oriented system for LoRaWAN, conceived to preserve the identity of endpoints and data they exchanged in the network. PIVOT analyzes the LoRa RF traffic stream to identify devices exposed to privacy threats. Its detection procedure works with the pattern matching principles and runs using a real-time algorithm. In this way, PIOVT can notify the network in the shortest possible time about the presence of vulnerable devices. The measurements it outputs, such as the number of devices detected, can give the operator a view of the current safety state of the network, supporting it in applying immediate countermeasures.

1.2 Structure of the thesis

The rest of this thesis is structured as follows:

1. Chapter 2 overviews the LoRaWAN technology and its features, focusing on the aspects of this protocol associated with the objectives of this thesis.
2. Chapter 3 discusses security issues that concern LPWAN technologies and, in detail, the privacy-related vulnerabilities that affect the LoRaWAN protocol and their implications for the customers.

3. Chapter 4 introduces PIVOT, first defining the design goals, then explaining its core algorithm and the metrics applied.
4. In Chapter 5 we explain the various countermeasures that the operator, based on the PIVOT output, can enforce to make the network securer.
5. Chapter 6 illustrates the implementation and the results of our experimental studies.
6. Chapter 7 discusses conclusions and possible directions for future work.

Chapter 2

LoRaWAN

Today IoT devices use several different technologies to support their communications, but not all of them are suitable for modern applications. Well-known wireless solutions, such as Wi-Fi or Bluetooth, require a lot of energy to send data and have a limited transmission range. LoRaWAN protocol is ideal for applications requiring long-range or deep in-building communication among many battery-operated sensors since they have low power requirements and need to collect only small volumes of data. Since LoRaWAN packets require a minimal amount of energy to be sent and are transmitted a few times a day, the devices' battery can last for many years. Moreover, the network can support millions of messages. For example, a single eight-channel gateway carries hundred of thousand of messages daily and when applications require more capacity, it's enough to add additional gateways to the network.

To better comprehend the LoRaWAN protocol, we start by describing the technology stack. As shown in Figure 2.1, LoRa, which represents the physical (PHY) layer, is the wireless modulation used to create the long-range communication link. On the other side, LoRaWAN represents the MAC layer and is the networking protocol that provides services such as bi-directional communication, mobility, and localization services.
In the following sections, first we brief analyze the key characteristics of the LoRa physical layer, then we detail LoRaWAN, describing its network architecture, the devices format, and the different activation methods.

2.1 Introduction to LoRa

LoRa, which stands for Long Range Radio, is an RF modulation technology for Low-Power Wide Area Networks (LPWANs). It is mainly targeted for M2M and IoT

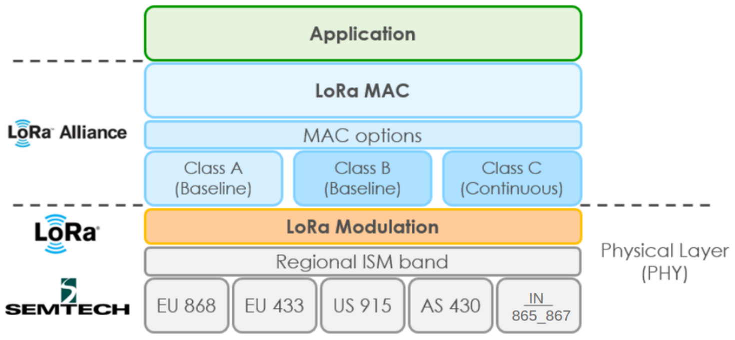


Figure 2.1. LoRa technology model

networks and enables public or multi-tenant networks to connect several applications running on the same network. Created by the French company Semtech [21] to standardize LPWANs, LoRa grants long-range transmissions, going from five kilometers in urban areas to 15 kilometers in rural areas [24].
In detail, LoRa is a *spread-spectrum* modulation technique. It derives from the existing Chirp Spread Spectrum (CSS) and operates in a fixed-bandwidth channel of either 125 kHz or 500 kHz for uplink channels and 500 kHz for downlink channels. The LoRa modulation ensures a trade-off between sensitivity and data rate.
Furthermore, LoRa uses orthogonal spreading factors to preserve the battery life of connected sensors by making adaptive optimizations of nodes' power levels and data rates.

2.1.1 Key LoRa Modulation Properties

These are the essential configuration parameters of LoRa radio

- **Carrier Frequency** (CF): It is the frequency used for the communications from node to the gateway. LoRa operates at unlicensed frequency ISM bands 863-870 MHz in Europe and 915 MHz in U.S. [16].
- **Spreading Factory** (SF): It is the number of chirps per symbol. As reported in Table 2.1, LoRa has 6 SF, whereas higher SFs allow larger coverage areas and, one symbol has 2 SF chirps for the overall frequency.

Table 2.1. Sensitivity of LoRa Receiver

BW	**Spreading Factor**					
	SF7	*SF8*	*SF9*	*SF10*	*SF11*	*SF12*
125 kHz	-126.50	-127.25	-131.25	-132.75	-134.50	133.25
250 kHz	-124.25	-126.75	-128.25	-130.25	-132.75	-132.25
500 kHz	-120.75	-124.00	-127.50	-128.75	-128.75	-133.25

- **Bandwidth** (BW): LoRa has three bandwidth options: 125 kHz, 250 kHz, and 500 kHz. The 125 kHz bandwidth is used for the 863-870 MHz frequency band, the 500 kHz bandwidth for rapid transmissions, and the 125 kHz bandwidth to cover wide areas. The correlation among duration of symbol T_s, the bandwidth BW, and the spreading factor SF is given by:

$$T_s = \frac{2^{SF}}{BW}$$

- **Coding Rate** (CR): LoRa adds a forward error correction (FEC) in every data transmission by encoding 4-bit data with redundancies into 5-bit, 6-bit, 7-bit, or even 8-bit. With this redundancy. the LoRa signal can endure short interferences. The Coding Rate (CR) value needs to be adjusted according to the conditions of the channel used for data transmission. In case of many interferences in the channel, it's recommended to increase the value of CR, even if it also increases the duration of the transmission. Coding rate expression is $CR = \frac{4}{4+n}$, where $n \in \{1, 2, 3, 4\}$. The modulation bit rate R_b is defined as:

$$R_b = \frac{4}{4+n}\frac{BW}{2^{SF}}$$

These parameters impact the receiver sensitivity. As the bandwidth increases, the sensitivity of the decoder lowers. The decrease in code rate helps to reduce Packet Error Rate (PER) against interference. For example, a message transmitted with a 4/8 code rate is more resilient on channel implications than a message with a code rate of 4/5.

2.2 LoRaWAN network topology

Typically, IoT applications adopt a mesh architecture to increase communication ranges and cell sizes of the network. However, this approach can affect the device's battery life, as the nodes have to forward messages to each other [13]. Conversely, to increase sensors' lifetime, the LoRaWAN network uses the simpler star-of-stars topology. This architecture consists of the following elements:

- **End Devices (EDs)**: low power sensors/actuators, battery operated. They are wirelessly connected to the network through gateways and use the LoRa RF modulation to communicate.

- **Gateway (GW)**: It receives messages from end devices and forwards them to the Network Server. A gateway connects to the Internet through an IP backbone.

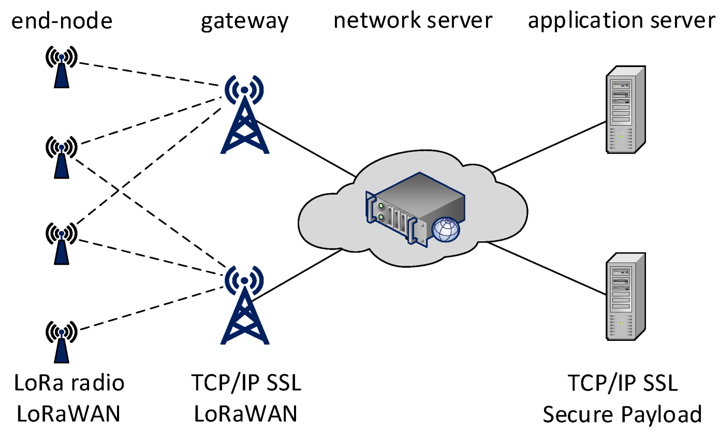


Figure 2.2. LoRaWAN architecture

- **Network Server (NS)**: This server manages the entire network. It receives IP traffic from gateways and is responsible for network management.
- **Application Server (AS)**: The application server processes messages received from end devices and generates downlink payloads to send to end devices. There may be more than one Application Server on the network

Since LoRaWAN networks use an ALOHA-based protocol, end devices don't need to be associated with a specific gateway for accessing the network. Consequently, the gateway acts simply like relays to which endpoints transmit data via the LoRa RF interface. On the other side, gateways forward the received packets to servers using the common IP networks such as 3G/4G, Ethernet, or WiFi.

2.3 Device Classes

LoRaWAN specifies three classes of devices: A, B, and C. These classes are distinguished by each other by the MAC procedures and by power consumption profiles. All devices implement Class A, whereas Class B and Class C extend the specification of Class A devices. Figures 2.3 illustrate the three classes, defined as follow:

1. **Class A** Class A devices support bi-directional communication. Both uplink and downlink transmission are done in the same randomly chosen radio channel. Every node observes the acknowledgment in receive windows, RX1 and RX2, during downlink transmission. Delays from the end of an uplink transmission to the start of receive windows (RX1 and RX2) are defined as Receive Delay 1 (RX1Delay) and Receive Delay 2 (RX2Delay, or RX1 + 1s).

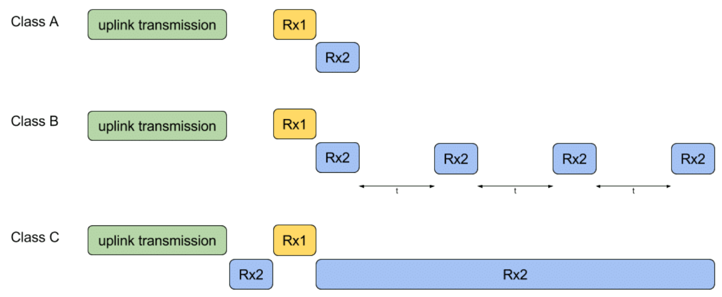


Figure 2.3. The classes of LoRaWAN devices

2. **Class B** Class B devices can open extra receiving slots at the scheduled times. The gateway generates a ping slot to integrate end devices to receive additional windows and broadcasts a time-synchronized periodic beacon to allow the Network Server (NS) to be aware of the listening status of end devices. Class B devices distribute the radio channel for downlink and uplink transmission to overcome the collision effect.

3. **Class C**: Devices operating in Class C have the receive windows almost always open (they close the window only when they transmit). For this reason, Class C devices require more power to operate than Class A or Class B. In turn, they have the lowest latency for communication.

2.4 Frame format

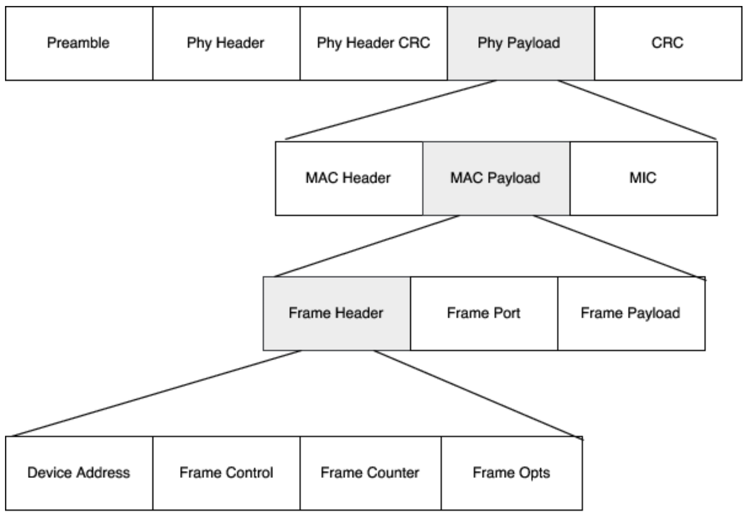


Figure 2.4

The LoRaWAN frame, shown in 2.4, is structured as follows:

1. *Physical Layer.* It has a preamble, that defines the modulation scheme, followed by a header and a payload. The header contains information such as payload length and whether the Payload 16-bit CRC is present in the frame (only uplink frames contain payload CRC). The payload contains the MAC frame.

2. *MAC Layer.* It consists of a header, a payload, and a Message Integrity Code (MIC). The header defines protocol version and message type, the payload contains the application data, and the MIC, computed using the header and the payload portions, is used to prevent the forgery of messages and authenticate the end node.

3. *Application Layer.* It consists of a header, a port, which value depends on the application type, and the payload. The header contains information about the device and the network and commands used to change parameters such as the data rate or the transmission payload. The payload contains the transmitted data and is encrypted with the AES 128 algorithm.

2.4.1 Message types

LoRaWAN message types transport MAC commands and application data. These messages are categorized into two main families, uplink and downlink messages, based on the direction they travel [20]:

1. Uplink messages are sent by ED to the Network Server, relayed by one or many gateways.

2. Each downlink message is sent by the Network Server to one ED and is relayed by a single gateway.

LoRaWAN v1.1 defines several MAC message types, presented in the Table 2.2. These messages are identified by the first bytes of the LoRaWAN packet, in the MHDR field, and specifically in a section of three bites, called *MType*.

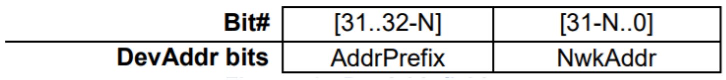
Table 2.2. MAC message types that can be found in LoRaWAN 1.1

MType	Description
000	Join Request
001	Join Accept
010	Unconfirmed Data Up
011	Unconfirmed Data Down
100	Confirmed Data Up
101	Confirmed Data Down
110	Rejoin-request
111	Proprietary

1. Join Request, Join Accept and Re-Join are three messages used to establish a connection between the LoRa End Device and Network Server.
2. Confirmed Data are data messages that need to be acknowledged by the receiver.
3. Unconfirmed Data are data messages that do not require any acknowledgment.
4. Proprietary messages are used to incorporate non-standard message format functionalities.

Addressing

Each device has a global address, the *DevEUI.* It is a unique EUI64 identifier supplied by the manufacturer, stored in non-volatile memory, and used as a MAC address. The first 3 bits represent the Organizationally Unique Identifier (OUI), purchased from IEEE that represents the constructor.



Bit#	[31..32-N]	[31-N..0]
DevAddr bits	AddrPrefix	NwkAddr

Figure 2.5. DevAddr fields

In addition, the LoRaWAN specification defines the *DevAddress*, a dynamic 32-bit temporary address generated directly by the Network Server. As shown in Figure 2.5, The first significant bits represent the *AddrPrefix.* It derives from NetID, the NS unique identifier attributed directly by the LoRa-Alliance to network providers and used in routing messages in the correct network. AddrPrefix describes the Network Server that is currently managing the device. The least significant bits define the *NwkAddr*, the network address of the ED, set by the network manager. Network Servers know for each device the unique DevEUI and the current DevAddress.

2.5 Device Activation

A LoRa device, to send and receive messages in the network, needs to be registered in the network, following a procedure called *activation.* Before the activation, three parameters of the device should be configured: the *AppEUI*, the *DevEUI*, and *AppKey* (an AES-128 bit secret key known as the root key). The activation procedure generates the *DevAddress* and two session keys, *nwkSKey* and *appSKey*, used to encrypt and transmit packets to the server. LoRaWAN offers two different activation methods: Over-the-Air Activation (OTAA) and Activation by Personalization (ABP).

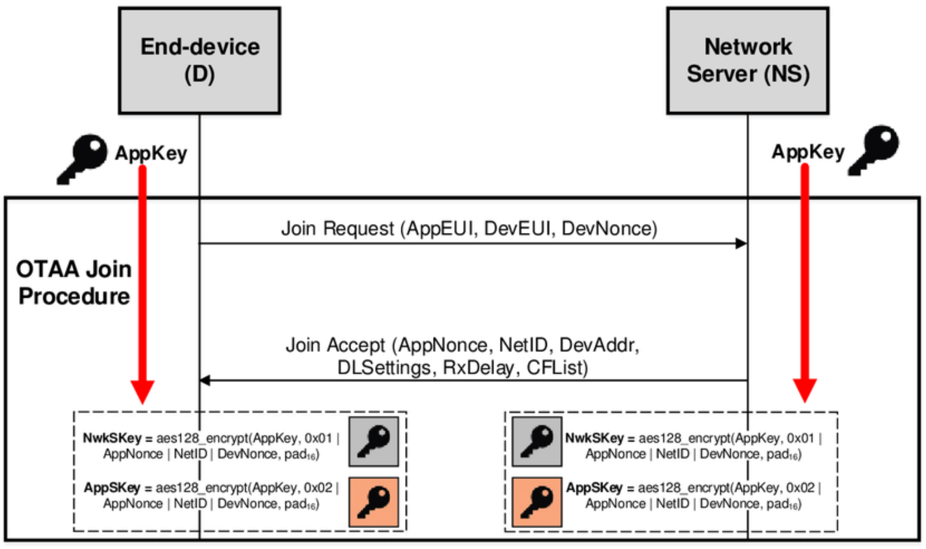

End-device (D)
Network Server (NS)
AppKey
AppKey
OTAA Join Procedure
Join Request (AppEUI, DevEUI, DevNonce)
Join Accept (AppNonce, NetID, DevAddr, DLSettings, RxDelay, CFList)
NwkSKey = aes128_encrypt(AppKey, 0x01 | AppNonce | NetID | DevNonce, pad16)
AppSKey = aes128_encrypt(AppKey, 0x02 | AppNonce | NetID | DevNonce, pad16)
NwkSKey = aes128_encrypt(AppKey, 0x01 | AppNonce | NetID | DevNonce, pad16)
AppSKey = aes128_encrypt(AppKey, 0x02 | AppNonce | NetID | DevNonce, pad16)

Figure 2.6. OTAA message flow

Since OTAA is more securer and more flexible than ABP, it is the recommended process for activation for LoRaWAN [12].

2.5.1 Over-the-Air Activation

Devices perform a join procedure with the network to receive a dynamic device address and negotiate security keys. This procedure requires the exchanging of two MAC messages between the end device and the Network Server:

- *Join-request*, from the ED to the Network Server.
- *Join-accept*, from the Network Server to the ED.

The end device initializes the join procedure, which sends to the network the Join-request message, composed of the following fields: *JoinEUI*, an IEEE EUI64 address that uniquely identifies the Join Server, the *DevNonce*, a 2-byte counter, used to prevent replay attacks, and the *DevEUI* global identifier. The Network Server processes the request message and generates the Join-accept message. The main fields of the Join-accept are the *NetID*, a network identifier, the *JoinNonce*, a device-specific counter value used to calculate the session keys, and finally, the *DevAddr*. The Join-accept message, encrypted using AES in ECB mode, is sent to the end device, now allowed to register in the network.

2.5.2 Activation by Personalization

Activation By Personalization (ABP) directly ties an end device to a pre-selected network. Devices and Network Server stores directly the DevAddr and the two session keys NwkSKey and AppSKey instead of the DevEUI, AppEUI, and the AppKey (each end device should have a unique set of NwkSKey and AppSkey). Although this activation procedure is easier than OTAA, it is less secure. To switch the network, devices have to manually change the keys. Moreover, they keep the same security session for their entire lifetime.

Chapter 3

Threat model

In this chapter, we deepen several privacy weaknesses that affect the current version of the LoRaWAN protocol. First, we introduce the security policies adopted in designing LPWAN technologies. Next, we explain the security mechanisms of LoRaWAN and their flaws. Finally, analyzing the privacy-related problems of this protocol, we identify the object of study of this thesis, a threat associated with the analysis of two identifiers used in LoRaWAN devices.

3.1 Security requirements

Like other IoT technologies, the nature of the information produced, processed, transmitted in LPWANs has significant implications on network security. Confidentiality, Integrity, and Availability, otherwise known as the *CIA* triad, are the three main security principles followed in developing the security of the LPWAN technologies [1]:

CONFIDENTIALITY
INTEGRITY
AVAILABILITY

Figure 3.1. The CIA triad

1. *Confidentiality.* It refers to a situation whereby the transmitted data are

Table 3.1. Common attack types in the CIA triad and their implications

Attack type	Attack features	Implications
Eavesdropping	Overhear and intercept data	Gain access to sensitive/private information
Jamming	Intentional transmission to disrupt communication	Cut-off communication, causing congestion, exhausting energy
Replay attack	Repeat a valid data transmission	Generate false messages, increase congestion
Wormhole	Create low latency tunnel between two malicious nodes	Sending false or out-dated data
Man-in-the-middle	Sniff network to intercept communication between nodes	Gain access to sensitive or private information
Denial of service	A general attack type that can include multiple attacks happening simultaneously	Disrupt normal operation of the network

accessible only to the sender and the recipient. There are several ways to compromise the confidentiality, such as the well-known Man-In-The-Middle (MITM) technique, where the attacker furtively alters the communications between two nodes.

2. *Integrity.* Data integrity is what the *I* in CIA Triad stands for. It refers to the protection of network data, preventing unintentional or intentional alteration of the information by unauthorized intruders. Typical integrity attacks performed on LPWANs are the replay attack and the wormhole attack [9]. The replay attack happens when a valid packet transmission is maliciously manipulated. The wormhole attack consists of a situation whereby an intruder receives network packets at one location, forwards them to another node, and continues to replay the packets within the entire network. Often the wormhole attack is be performed using two devices, the sniffer and the jammer. While the sniffers capture the network packets, the jammer blocks or interference with the communications.

3. *Availability.* This is the final component of the CIA Triad and refers to the actual availability of data. Network resources should be always obtainable for the authorized user when required. This requirement prevents bottleneck situations that hinder information flow. The most famous availability threat is Denial of Service (DoS) attacks, the attempt to consume network resources or bandwidth [3]. Common DoS-related attacks include DNS flood, Internet Control Message Protocol (ICMP) broadcast, and SYN flood.

3.1.1 Security in LoRaWAN

Since devices are designed to last for many years, security plays a key role in the LoRaWAN protocol. The security design criteria follow the CIA triad and adopt the main LoRaWAN principles: low power consumption, low implementation complexity, and low-cost [18]. The fundamental properties are:

1. Mutual authentication.

2. Integrity protection.

3. Confidentiality.

Mutual authentication, established between an ED and the network, ensures that only authorized devices can join authentic networks while MAC and application messaging are integrity-protected with two session keys, AppSKey and NwkSKey. As shown in Figure 3.2, each payload, encrypted by AES-CTR, carries a frame counter to avoid packet replay and Message Integrity Code (MIC) to avoid packet tampering. This protection, combined with mutual authentication, ensures that network traffic is not altered, comes from a legitimate device, and is not comprehensible to eavesdroppers.

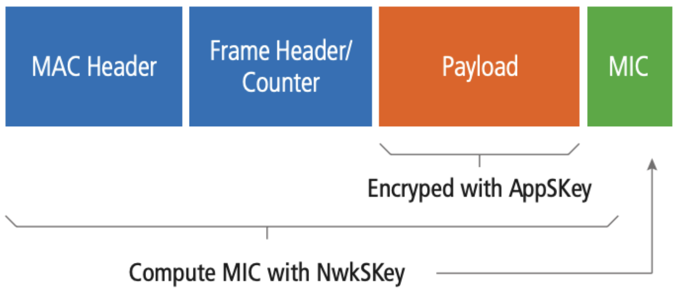


Figure 3.2. The encryption in LoRaWAN

Despite LoRaWAN being designed to be very secure with authentication and encryption mandatory, the 1.0 specification suffers from several security vulnerabilities, partially solved in the LoRaWAN 1.1. [22]. The newest version of the protocol mainly combat confidentiality issues by making specification changes to the MAC layer but needs to be made to resolve integrity, availability, and authentication vulnerabilities. For example, LoRaWAN lacks in creating a secure channel between the Node Server and the Join Server. In addition, the PHY layer is prone to Denial-of-service attacks due to impersonation and jamming attacks [23].

3.2 Privacy threats

As opposed to security, privacy aspects in LPWANs have received little attention. As shown in 3.3, since the long communication range of LPWAN technologies allows messages to be received several kilometers away, an eavesdropper can easily record sensitive information, even if located many hundred meters from endpoints. By leveraging features of the raw wireless signal, attackers can interfere with the

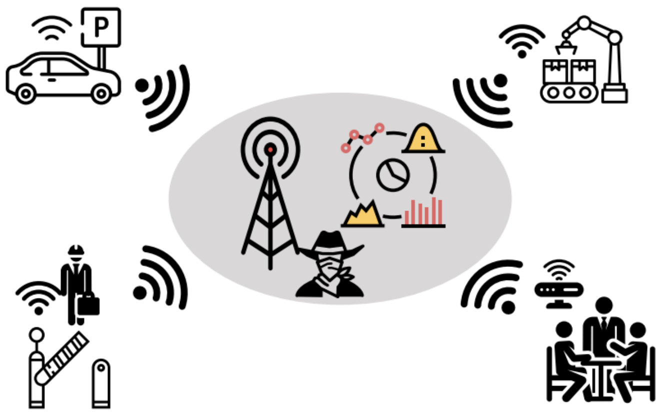


Figure 3.3. When end devices transmit messages to the server, the passive adversary can easily collect sensitive information.

activities of devices operating in the network. In detail, we assume that the passive adversary, equipped with one or more receivers collecting end device transmissions, is within the communication range of at least one gateway and that it has a *priori* knowledge about the application associated with an ED.
With these assumptions, the intruder can perform two relevant privacy- related threats.

1. **Fingerprinting**. Device Fingerprinting (DFP) denotes the set of techniques used to identify a device using the information extrapolated from the packets it uses to communicate over the network [14]. DFP is commonly used in general-purpose devices to track user behavior and application usage. Interesting implementations include browser fingerprinting for web analytics, fraud detection, and accountability [15]. Despite the benefits for the user experience, DFP poses also security and privacy risks. For example, LoRa devices can be uniquely recognized using physical (PHY) layer fingerprinting [25]. It leverages on differences in the analog RF signals sent by wireless devices, caused by imperfections introduced in the analog hardware components during the manufacturing process [7], demonstrating that an adversary can identify a transmitter independently of the modulation scheme or cryptographic mechanisms used.

2. **Information leakages**. Since LPWAN is technology-constrained, attackers can easily reconstruct a view of the network they monitor. While other wireless devices transmit messages not necessarily related to real situations, LPWAN sensors are simpler and often dedicated to no more than one function, such as transmitting data when a given event occurs [17]. Moreover, the LPWAN event

space is small and binary, making the purpose of the individual sensors even clearer. Just consider the case of a parking system application that notifies clients if a given parking lot is occupied or not [5]. Endpoints send only two binary messages, the lot is empty or the lot is not empty, and only when there is a change of state (a vehicle left the lot). In absence of obfuscation techniques, a potential attacker recognizes the presence of real messages by simply eavesdropping on the LPWAN channel.

In other scenarios, a deviation from standard transmission behavior presupposes the occurrence of an event. For example, in the case of an application that monitors the availability of a meeting room, the LoRa sensor transmits less when there is a series of long meetings that keep the room busy. The eavesdropper, collecting the messages in a given time frame, can use *traffic analysis* techniques to extrapolate the transmission behavior and gain information about given circumstances.

3.2.1 Open issues

Even if the privacy aspects in LoRaWAN are not a new research topic, there are still open and unexplored problems in this context. As we have seen, LoRaWAN sensors are configured to have a very minimal behavior, causing common vulnerabilities, such as fingerprinting and information leakages. Despite the various countermeasures proposed and applied, there are still open issues that cause privacy-related matters. In this work, we focus on a threat correlated to two LoRaWAN identifiers, the *DevAddr* and the *DevEUI*.

Re-identififying addresses

To avoid malicious fingerprinting, LoRaWAN doesn't expose to eavesdroppers sensitive information concerning devices and the network to which they are associated. For a given endpoint, only the application and the network knew the association between the current DevAddress and the DevEUI. Since DevEUI is revealed only during the OTAA procedure and the DevAddress is sent only in uplink messages, an ED can't send a packet with both identifiers exposed.

The decoupling of the DevEUI from the DevAddress guarantees devices' anonymity. Indeed, the NwkID field contained in the DevAddress reveals information about the network, and the OUI field of DevEUI contains information about the constructor of the device. By combining these two elements, the attacker increases its knowledge about the targeted LoRaWAN network. For example, it can determine which devices from a given operator (OUI) are around an assigned gateway (NwkID) or if the network belongs to private or experimental projects. Several studies demonstrate

that the current LoRaWAN specification is not strong enough to assure that the association of these addresses remains unknown to unauthorized parts. For example, a technique proposed in [4] recognizes the DevAddress-DevEUI association relying on the timing between a Join-Request and the immediately following uplink message. The reidentification of these identifiers allows the discovery of peculiar activities in the network. By analyzing the traffic flow of Bouygues Telecom, this research demonstrates that during a peak in device activations, the newly joined devices are used only for test purposes.
On the opposite, a second research [28] goes further: it underlines that this temporal linking is not always constant and proposes a pattern-based approach, named DEVIL, which, tested on a real dataset provided by an Italian operator, declares more accurate results. DEVIL is a batch-type algorithm: it processes all the data a posteriori, collecting uplink and join request messages in a given time window and elaborating them later.

As we have seen, pattern matching proves to be an effective strategy in detecting those devices whose pair *DevAddr* - *DevEUI* of identifiers can be easily recognized. In this thesis, we propose a defensive strategy that, exploiting the advantages of pattern matching, can prevent the re-identification attacks mentioned above.

Chapter 4

The privacy monitoring system

As demonstrated in the previous chapter, although the LoRaWAN protocol is designed to be very secure, some still unsolved privacy issues can compromise the confidentiality of information exchanged within the LoRa network. In this chapter, we introduce *PIVOT*, a LoRaWAN privacy monitoring system that, analyzing the traffic flow, detects any weak endpoint that could expose sensitive information to unauthorized parties. The goal of PIVOT is to strengthen the traffic exchanged between ED and servers to avoid adversaries from conducting privacy-related threats such as malicious fingerprints or information leaks.
The rest of the chapter is structured as follows. First, in Section 4.1, we present the main principle and guidelines used in designing our system. In Section 4.2, we deepen PIVOT, going from the assumptions to the definition of the detection algorithm. Finally, we illustrate the core metrics supplied by PIVOT that the operator can use to monitor the network environment and check the current status of devices.

4.1 Design Goals

The following are the main points followed for the development of PIVOT.

1. *Detection* of vulnerable devices. To avoid the threat described in Section 3.2.1, PIVOT should properly identify that endpoints of the network whose *DevAddr* - *DevEUI* address pair could be discovered by unauthorized listeners.

2. *Monitoring* the status of end-device. Our system should give the operator a transparent vision of the state of the network, showing, for example, the number of detected devices.

3. *Strengthening* of the network. PIVOT should represent an additional security layer for the LoRaWAN protocol to better preserve the confidentiality of the

LoRa traffic exchanged between the endpoints and the Network Server.

4. *Modelling* the behavior of periodic devices. Since EDs that send packets regularly are inclined to threat we are trying to avoid, PIVOT should discover and model their behavior.

5. *Stream* of data. PIVOT should constantly analyze the traffic. Since it cannot analyze the entire group of messages only once they have all been sent, it has to make a decision every time a single message is exchanged.

6. *Minimize* the delay. Vulnerable devices need to be detected quickly to leave them exposed as little time as possible.

7. *Accuracy* of the detection. The detection procedure of PIVOT should be detailed and precise as far as possible. The error rate should be kept low.

8. *Lightweight.* The system should be designed to have a small memory footprint (RAM usage) and low CPU usage, overall a low usage of system resources.

9. *Filter* only the necessary data from all network traffic. As we will explain later, not all LoRa messages are relevant to our goal.

10. *Passivity.* Our system should collect and analyze the packets without interfering with the normal operations of the network.

11. *Scalability.* PIVOT should be able to scale, working well on both a small data set and a larger one.

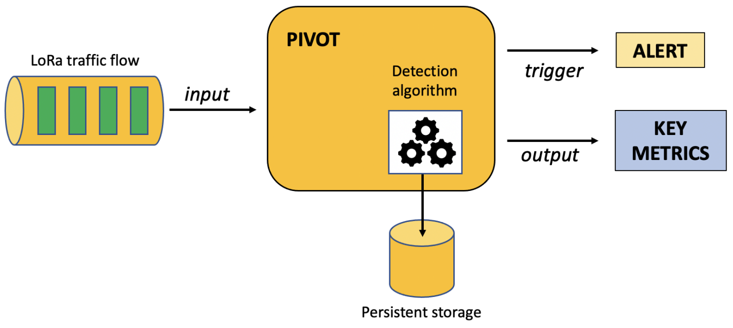


Figure 4.1. The architecture of PIVOT

4.2 System Description

The main policy of PIVOT is to *detect* and *protect* devices in the LoRa network that are vulnerable to information disclosure. In this section, we explain how our system can achieve this.

4.2.1 Assumptions

The endpoints of the LoRa network in which PIVOT operates should be LoRaWAN v1.1 compliant. If they get disconnected or lose the connection with the Network Server, should try a new activation procedure, which gives them a new DevAddress. Devices should perform exclusively the Over-the-Air Activation (OTAA) to join the network since PIVOT needs to intercept and collect *Join-request* messages used in the activation procedure. This condition results satisfied because LoRaWAN specification recommends using OTAA for security and scalability reasons.
Another assumption is that these devices should transmit packets periodically. These requirements find evidence in real case studies. For example, the analysis carried out on a LoRaWAN service of an Italian operator [28] shows up that the majority of devices send 24 packets each hour, the 25th one after 30 seconds, and then repeat the process. We can argue that when a device sends packets periodically, it follows a temporal *pattern*, a behavior repeated continuously over time.

4.2.2 Architecture

Figure 4.1 reports the high-level architecture of PIVOT. The system passively collects the LoRa stream, filtering *uplink* and *join-request* messages, the only needed for our

purpose.
Each elaborated frame represents the input of the so-called *detection* algorithm. This procedure, which is the core of PIVOT, analyzes the incoming messages and tries to deduce if the devices that generated them could be at risk and, in that case, it triggers an alert to the network operator.
During its analysis, the detection algorithm constantly outputs to the operator a set of metrics it can use to check the current network status. They consist of statistical reports about the current number of join/re-joined devices and correlated percentage of vulnerable endpoints that are still operating. In this way, it is possible to check the current level of security of the network and, if needed, execute actions to decrease the level of exposure of vulnerable EDs.
Finally, PIVOT uses a persistent storage system to store some relevant data used to support the functioning of the detection algorithm.

From a technical point of view, this algorithm belongs to the streaming algorithms family, which main characteristics are:

1. It processes a *stream* of data (the LoRa RF traffic).
2. The input, presented as a sequence of items, can be examined in only a few passes.
3. The memory access and the processing time are limited.

As shown in Figure 4.2, our system eavesdrops on the whole LoRa radio link area. Since symmetric key encryption is applied only to the payload, PIVOT has complete access to the text-clear header of the frames it collects. In particular, it requires the fields reported in Table 4.1.

Table 4.1. Key fields in the header, used by PIVOT

PARAMETER	**DESCRIPTION**
DEV_ADDR	Unique identifier of the ED in the network
DEV_EUI	Unique identifier of the physical ED
FCnt	Frame counter
TMST	Internal clock timestamp
TYPE	Type of packet

When the detection algorithm receives as input *uplink* frame with a DevAddress a it has never seen before, checks which of the following two circumstances is met:

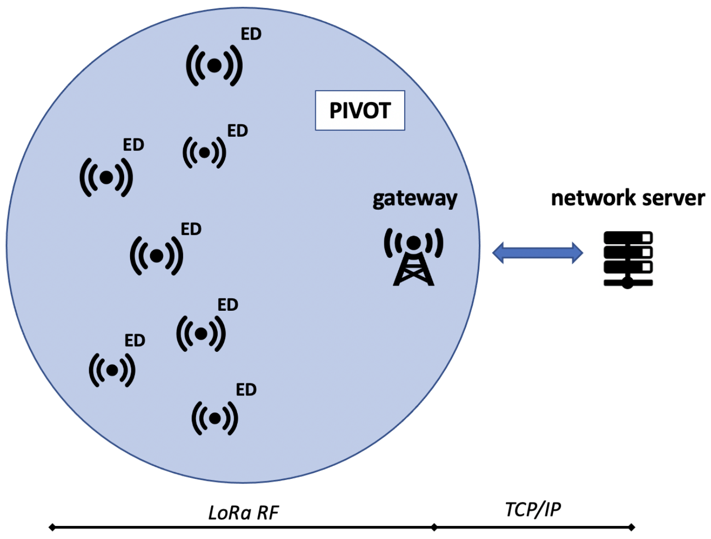


Figure 4.2. The coverage area of PIVOT

1. The DevAddress a is linked to an ED that joins the network for the first time.
2. The DevAddress a is linked to an ED, already in the network, that has disconnected and performed a new join procedure.

When the second scenario occurs, PIVOT recognizes that the ED has a too predictable behavior, which could be the object of attack. It triggers the alert to the operator that, modifying the parameters of the device, can prevent it from being identified again.

4.2.3 Patterns

By hypothesis, each device has a transmission frequency, i.e. it regularly sends data to the server. Therefore, even when it disconnects from the network to then re-join it later, its behavior does not change. Formally, when an ED transmits packets periodically, we suppose it follows a temporal *pattern* of period τ that repeats continuously over time. Then, two DevAddress a_1 and a_2 which belongs to the same end-device, can be linked using a *pattern-matching* strategy.

Notation

PIVOT focuses on the time elapsed between an uplink message and the next one from the same endpoint. In detail, let t_i the timestamp TMST of a packet p with a frame counter (FCnt) $p_{fcnt} = i$, we define the *segment* as the temporal distance between two consecutive messages with the same DevAddress (DEV_ADDR):

$$s_i = t_{i+1} - t_i$$

Consequently, we define the pattern P of the device with DevAddress a as a *chain* of segments.

$$P_a = \{s_1, s_2, ...\}$$

The pattern has period $\tau = s_1 + ... + s_n$ where n is the number of segments in the chain of P.

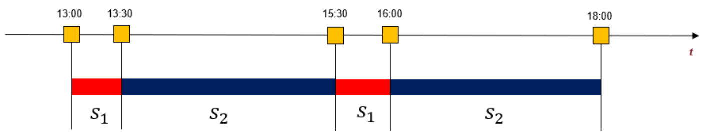


Figure 4.3. An example of pattern $P = \{s_1, s_2\}$

The Figure 4.3 reports an example of pattern P. This end device sends uplink packets with the following frequency:

1. Send the first packet of the sequence.
2. Waits half an hour, then sends the second packet of the sequence.
3. Waits two hours, then sends the third packet of the sequence.
4. Repeat the sequence.

Let $s_1 = 0.5$ (half an hour) and $s_2 = 2.0$ (two hours), then the device has the following pattern:

$$P = \{s_1, s_2\} = \{0.5, 2.0\}$$

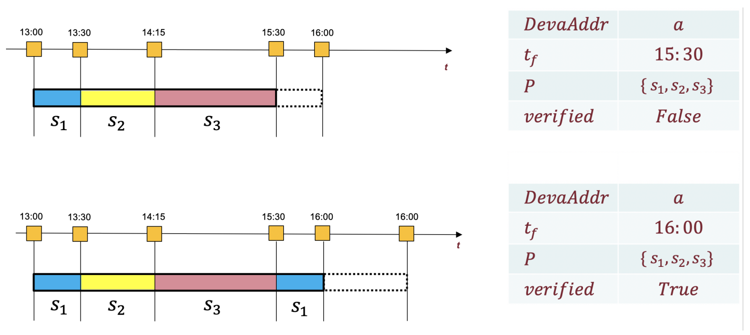
with period $\tau = s_1 + s_2 = 2.5$. In other words, this device repeats its pattern every two and a half hours.

Building and updating patterns

PIVOT keeps track of every initialized pattern P by storing the following dynamic data:

- The chain of segment $\{s_1, ..., s_n\}$ that composes the pattern P.
- The current DevAddress a of the device.
- The timestamp t_f of the last packet sent by the device.
- A flag *verified*, initially set to `False`.

For each new DevAddress a, PIVOT initialize an *empty* pattern $P_a = \{\}$. The goal to fill the chain of P_a, discovering the number and size of the segments that compose it. The last timestamp t_f received allow us to calculate the *current* segment each time a new packet arrives and, if necessary, add it to the chain.
The *verified* is a boolean flag that PIVOT sets to *True* when it can estimate with a significantly high probability that the chain is complete. For example, the flag $verified$ associate to the pattern $P_a = \{s_1, s_2, s_3\}$ is set to *False* until the algorithm has correctly calculated all three segments of the chain.



DevaAddr	*a*
t_f	15:30
P	$\{s_1, s_2, s_3\}$
verified	*False*

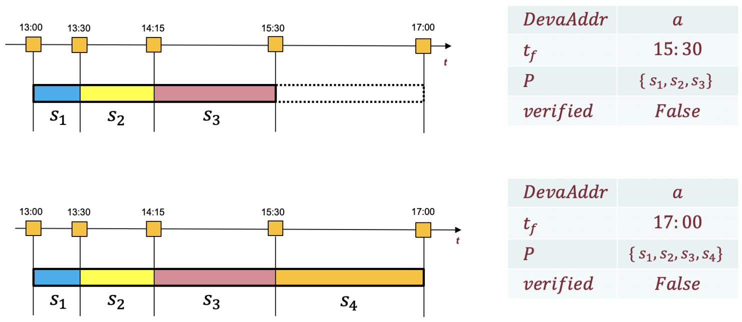
DevaAddr	*a*
t_f	16:00
P	$\{s_1, s_2, s_3\}$
verified	*True*

Figure 4.4. Updating of a pattern with a chain of three segments. After the update PIVOT changes only the parameters t_f and $verified$. The chain don't change.

When PIVOT receives a new packet p with DevAddress a, it triggers the *update* subroutine, reported with an example in figure 4.4. It works as follow:

1. It calculates the *current* segment $s = t - t_f$, where t is the timestamp of p and t_f is the last timestamp associated to a
2. It updates the last timestamp $t_f = t$.
3. It cheks if s is in the chain of P.
4. It updates, if needed, the flag $verified$.

If detail, if the segment s in not in the chain, the subroutine add it to the bottom of the chain and sets the flag $verified = False$. On contrary, if the segment is already in the chain, PIVOT gets the flag *verified* and, if it is currently set to *False*, switch it to *True*.



DevaAddr	a
t_f	15:30
P	$\{s_1, s_2, s_3\}$
verified	*False*

DevaAddr	a
t_f	17:00
P	$\{s_1, s_2, s_3, s_4\}$
verified	*False*

Figure 4.5. Updating of a pattern with a chain of four segments. After the update PIVOT changes the parameters t_f and append the new segment in the chain. The flag $verified$ remains set to False.

The figure 4.5 shows an example in which the chain of P_a remains still incomplete after the update subroutine. Since the current segment was not present in the chain, it is appended at the bottom and the flag *verified* remains set to `False`. Consequently, PIVOT cannot yet determine if this pattern is complete.

Comparing the patterns

Given two addresses a_1 and a_2 such that $a_1 \neq a_2$ and their patterns P_{a_1} and P_{a_2} such that $P_{a_1} \equiv P_{a_2}$. PIVOT concludes that P_{a_1} and P_{a_2} are *equal* and that a_1 and a_2 actually belong to the same device. This procedure is called *matching*, and is the way the system identifies which devices are too exposed to external threats. Two pattern P_1 and P_2 are equal when the following conditions are respected:

1. The two chains are of the same length i.e. the patterns have the same period.
2. The two chains are composed of the same number of segments.
3. The two chains are composed of the same segments.
4. the segments are *sequentially* distributed in the same way within the chains.

The figures 4.6 and 4.7 reports two examples where the pattern don't match.

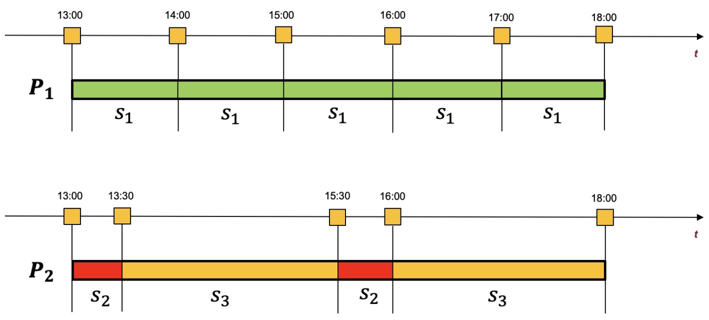


Figure 4.6. Example of patterns with different period, number and type of segments.

In the first exmple, the pattern P_1 consists of a single one-hour segment $s_1 = 1.0$, the pattern P_2 of two segments, one of 30 minutes $s_2 = 0.5$ and one of 2 hours $s_3 = 2.0$. PIVOT labels the two patterns $P_1 = \{s_1\}$ and $P_2 = \{s_2, s_3\}$ as different. First, they have two distinct periods $\tau_1 = 1.0$ and $\tau_2 = 2.5$. Second, the chains are composed by a diverse number and type of segments: the first chain one has one segment, the second one has two segments.

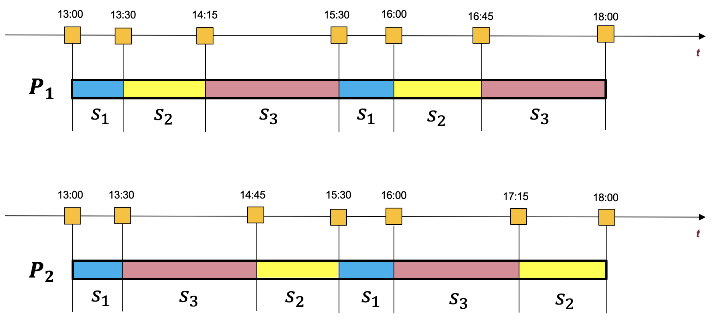


Figure 4.7. Example of different patterns with the same period, number, and type of segments.

In the second example, the patterns have some common components, but they still don't match. Despite they have the same periods $\tau_1 = \tau_2 = 2.5$, the same number and the same type of segments s_1, s_2, s_3, they are distributed differently in the two chains: $P_1 = \{s_1, s_2, s_3\} \not\equiv P_2 = \{s_1, s_3, s_2\}$.

4.2.4 Detection algorithm

The purpose of the detection algorithm is to detect potentially too exposed devices. It analyzes the LoRa traffic, keeps track of all the detected DevAddress, and uses a pattern-matching strategy to verify if two collected DevAddress could be associated with the same ED.

In detail, the detection algorithm uses a vector C to store all the identified patterns. It contains a series of elements P_a, where a is the current DevAddress associated to an ED, and $P_a = \{s_1, ..., s_n\}$ is the pattern the device follows. Since different patterns necessarily describe distinct EDs, in C, there cannot be two tuples denoting the same device. In other words, let E be the set of all the ED that currently join the network, we have:

$$length(C) \subseteq length(E)$$

The vector C allows checking if a given device, represented by its current DevAddres a, already exchanged messages in the network. When the detection algorithm reads a new packet with DevAddress a, it observe if the condition $P_a \in C$ occurs. If $P_a \notin C$ and this happens after the interception of a Join-request message, PIVOT cannot establish *a priori* the nature of the device associated to a. It initiates a pattern-matching process to compute if a belongs to a new device or a re-joined one, trying to achieve this task in the shortest time possible. In this way, in the case of potential matching, the algorithm minimizes the delay in triggering the alert.

Intercepting a Join-request message is a crucial part of the detection process. It permits PIVOT to comprehend which addresses it needs to investigate and which patterns we should use for the matching procedure. The pattern $P_a \notin C$, associated with an unknown address a, should *not* be compared with all the other patterns of C. We must exclude from our analysis all that pattern related to these two classes of addresses:

- All the addresses collected after the Join request, which have been noticed also before the Join-request. They are associated with devices that never disconnect from the network.
- All the other unknown addresses received after the Join-request, since a new device that joins the network has not more than one DevAddr.

The algorithm follows two sequential steps, called *Pre-Join* procedure and *Main* procedure. Figure 4.8 illustrates the two procedures. Let t_0 the timestamp of the first packet received by PIVOT after its activation and t_f the timestamp of the first Join-request packet. The Pre-Join procedure takes place in the time window $t_f - t_0$ and the Main procedure starts after the reception of the first Join-Request message, at t_f. In the next sections, we are going to examine these two steps in detail.

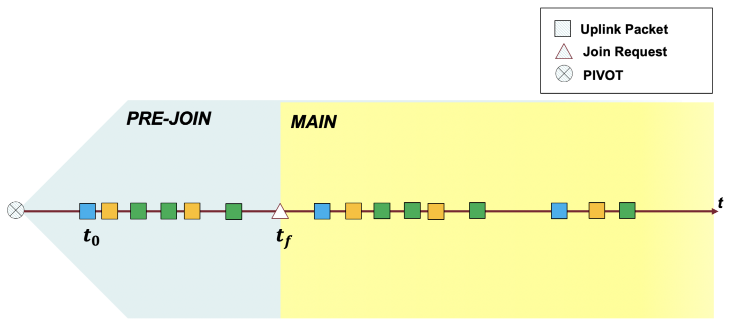


Figure 4.8. Representation of the time windows of the Pre-Join and Main procedures

Pre-Join procedure

The Pre-Join procedure is a necessary step to give to PIVOT an initial view about which EDs are currently operating in the network. Indeed, when the system is turned on and start receiving packets, it has no historical trace of the devices in the network, either of the messages they sent before its activation. This implies that the vector C is initially *empty* and therefore, there are no saved pattern to use in the matching process. Then, the Pre-Join procedure is a initial step through which the first patterns are identified and modeled. They will subsequently be used as a reference in the *Main* procedure.

In detail, for each packet p received in input, the algorithm reads the DevAddress a and the timestamp t. There are two scenarios:

1. $P_a \notin C$. A new pattern P_a is *initialized.*
2. $P_a \in C$. The pattern P_a is *updated*

According to the Section 4.2.3, the initialization of a new pattern consists in the creation of an empty chain $P_a = \{\}$ to store together with the DevAddress a, the timestamp t and the flag $verified$, set to *False*. In the same way, updating an existing pattern implies calculating the *current* segment $t - t_f$ and checking if s is already in the chain.

Algorithm 1 Pre-Join procedure

```
1: if $a$ in $C$ then
2:     $P \leftarrow C(a)$
3:     $P.update(t)$
4: else
5:     $P \leftarrow newPattern(a)$
6:     $C(a) \leftarrow P$
7: end if
```

The Pre-Join terminates when PIVOT receives as input the first Join-request. At the end of this phase, the number of DevAddress $a_1, a_2, ..., a_n$ collected matches with the number of devices that sent messages $ED_1, ED_2, ..., ED_n$. According to the LoRaWAN specification, a device ED_i modifies the DevAddress a_i only when it disconnects from the network and performs an new OOTA. This step ends exactly at the moment the first Join-request arrives. Then, for the entire duration of the Pre-Join procedure there is no ED that changes the associated DevAddress. All the pattern detected and stored in the vector C are in a *one-to-one* relationship with all the collected DevAddress.

As we have seen, a variable that determines the temporal window of the Pre-Join is the instant in which a device sends a Join-request. It is an uncertain and unpredictable phenomenon that can affect the accuracy of the detected patterns of C. Indeed, if the period τ of a pattern is longer than the duration of the procedure, the algorithm cannot correctly estimate all the segments that make up the chain. The flag *verified* is designed precisely to indicate the completeness of a pattern. Let V the list of patterns with the flag *verified* = *True*, then $V \subseteq C$. In the best case, at the end of the Pre-Join, the patterns in the vector C are all complete, allowing the detection algorithm to perform more precise matches.

The example reported in figure 4.9 shows two different patterns $P_1 = \{s_1\}$ and $P_2 = \{s_2, s_3\}$, where $s_1 = 0.5$, $s_2 = 0.33$ and $s_3 = 2.0$. If the duration of the Pre-Join procedure is an hour and three quarters (or *1.75*), the algorithm can't determine all the segments that composes the chain of P_2. Then to P_2 is associated the flag *verified* = *False*. On the contrary, since P_1 has a smaller period $\tau = 0.5$, is correctly recognized, the flag *verified* of P_1 is set to True.

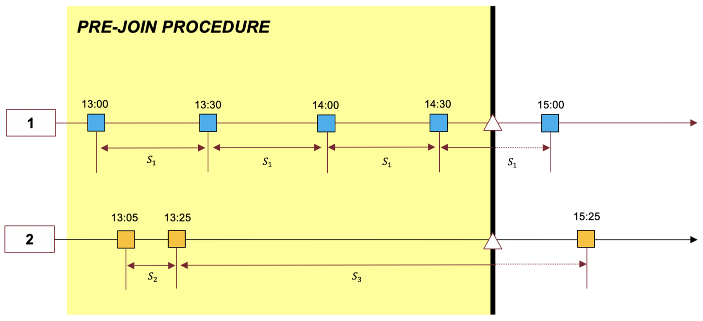


Figure 4.9. An example of pattern with different periods $\tau_1 = 0.5$ and $\tau_2 = 2.3$. At the end of the Pre-Join procedure, only the first pattern is recognized.

Main procedure

After the first Join-request message, the detection algorithm can no longer guarantee that a packet p with DevAddress a such that $P_a \notin C$ necessarily belongs to a new, unregistered device. This time we have to consider also the hypothesis that the address a belongs to a re-joined device. In detail, for each DevAddress a, we have:

1. $P_a \in C$. a refers to a known ED, and it associated to a known pattern.
2. $P_a \notin C$. a refers to an unknown ED. The algorithm must recognize if it is belongs to a new device or an old one that re-joined the network.

When the second scenario occurs, as in the Pre-Join procedure, a new pattern P_a is initialized. However, this time the pattern is not put in C, but in a new vector, called U, until the algorithm finds out if the address a represents a new device or not. Finally, the algorithm initialized the list $TA_a \leftarrow C$ of *potential* patterns which could match with P_a.

After the initialization of P_a, each time the system receives as input a new packet p with the same address a and a new timestamp t, the algorithm updates $P_a \leftarrow U$ and compares it with the patterns of TA_a. Initially, the $TA_a \leftarrow verified(C)$ i.e. is it is composed of all the elements of C whose patterns is complete (then, with *verified* flag set to `True`). After the updating of P_a, three situations can occur:

1. P_a doesn't match any element of TA_a.
2. The chain of P_a is a subset of one o more chains of the patterns of TA_a.
3. P_a matches *exactly* with one o more elements of TA_a.

When the first scenario occurs, the DevAddress a necessarily represents a new ED. TA_a is erased from the storage and P_a is removed from U to be appended in C.

On the contrary, in the second case the chain of segments of P_a matches with the *subchain* of one o more patterns of TA_a. This implies that the chain of P_a may still be incomplete. Then the algorithm left P_a in the vector U and remove from TA_a all that pattern with which P_a don't match.

In the last scenario, since there is a match between P_a and another pattern P_b, most likely the associated addresses a and b belong to the same device. The algorithm inserts the P_a in a reservated vector, called Q. When PIVOT will receive a new packet with DevAddress a such that $P_a \in Q$, the algorithm starts the *quarantine* subroutine, that will definitively confirm or reject the match between P_a and P_b.

Algorithm 2 Main procedure

```
1: if $P_a$ n $C$ then
2:     $P.update(t)$
3: else
4:     if $P_a$ in $U$ then
5:         $P.update(t)$
6:         if $P_a$ in $Q$ then
7:             quarantine procedure
8:         end if
9:         for all $P_{ta}$ in $TA(a)$ do
10:            if $match(P_a, P_{ta})$ then
11:                $Q \leftarrow P_a$
12:            else
13:                $TA(a).remove(P_{ta})$
14:            end if
15:        end for
16:        if $TA(a)$ is empty then
17:            new device
18:            $C \leftarrow P_a$
19:        end if
20:    else
21:        $P \leftarrow newPattern(a)$
22:        $U(a) \leftarrow P$
23:        $TA(a) \leftarrow verified(C)$
24:    end if
25: end if
```

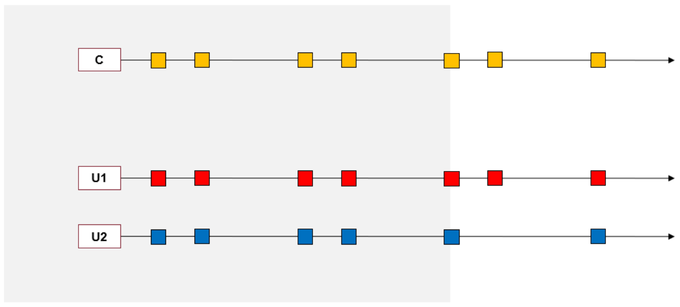


Figure 4.10. This example reports the two scenarios that can occur in the quarantine subroutine. In the first case, after updating the pattern $U1$, PIVOT confirms that $U1$ and C are linked to the same ED. In the second case, after updating the pattern $U2$, PIVOT confirms that $U2$ is associated to a new ED.

Quarantine

When the incoming packet p has a DevAddress a such that $P_a \in Q$, it means that P_a matches exactly with another pattern P_b. The objective of the quarantine subroutine is to confirm this match or, on contrary, to demonstrate that P_a and P_b are linked to different EDs. Given the timestamp t of p, the algorithm updates P_a and checks if P_b and P_a still match. If so, PIVOT triggers the `alert` the operator. If not, P_a belongs to a new device, then P_a moves from U to C. The figure 4.10 illustrates the two scenarios that can occur in the quarantine subroutine. Before of this step, the pattern C matches with both $U1$ and $U2$. In the first case, after updating $U1$ with the current timestamp t, the algorithm confirms that $U1$ and C still matches. In the second case, after updating $U2$, the algorithm concluded that it represents the pattern of another device.

Algorithm 3 Quarantine

```
1: $P_a \leftarrow U$
2: $P_b \leftarrow TA_a$
3: $P_a.update(t)$
4: **if** $match(P_a, P_b)$ **then**
5:     alert
6: **else**
7:     new device
8:     $C \leftarrow P_a$
9: **end if**
```

4.2.5 Metrics for operator

PIVOT supports the network operator in its routine operations, providing a series of statistical metrics that illustrate the current state of the administered network.

Number of Joins (NoJ)

The Number of Joins (NoJ) represents the overall *Join-request* messages intercepted and registered by PIVOT over the time. Generally, in LoRaWAN there is a *peak* of Join-requests only in the starting phase when the operator registers and activates all the devices that will operate in the network. All Join-requests following this peak denote the insertion of new devices or the reconnection of old ones, two uncommon events. A too high value of NoJ might indicate the malfunction of one or more devices that are disconnecting and reconnecting continuously from the network. As demonstrated, sending a Join-request message could involve the expose the addresses of the device to unauthorized parties, then operator should avoid that the EDs send too many messages of this type.

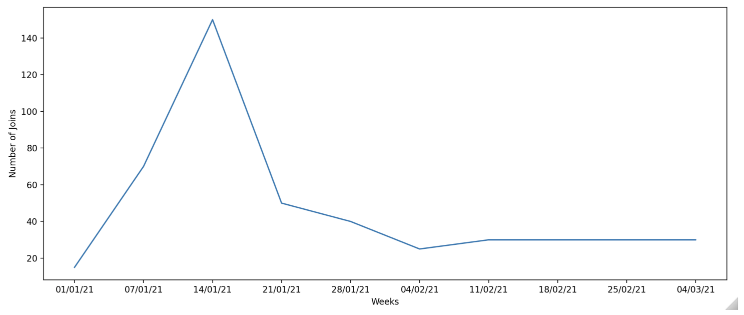


Figure 4.11. This plot reports an example of NoJ/weeks of a LoRaWAN network without anomaly devices. There is a peak only in the initial phase.

Number of Detected Devices (NoDD)

The Number of Detected Devices (NoDD) describes the current amount of devices classified by PIVOT as vulnerable. These EDs, that re-joined the network at least once, have a too predictable pattern that could disclose their sensible characteristics, such as the identifiers, to external eavesdroppers. NoDD is updated each time PIVOT triggers the alert to the operator. This metric depends on the *accuracy* of the detection algorithm of PIVOT, which may *misclassify* some devices or miss capturing all vulnerable ones.

Number of Unique Devices (NoUD)

The Number of Unique Devices (NoUD) is the number of ED registered by PIVOT. These devices have sent at least one message to the server, exposing their DevAdress. NoUD does *not* represent the total amount of devices of the network but only those that have transmitted packets since PIVOT was turned on. NoUD indicates the number of *currently active* devices.

Percentage of Detected Devices (PoDD)

The Percentage of Detected Devices (PoDD) is a value in the range [0, 1] and represents the ratio between *NoDD* and *NoUD*. It provides a clear view of the network, showing exactly how many devices are vulnerable among all those present. Since it is not unusual for LoRa devices to generate periodic traffic, the PoDD value is generally always greater than zero. Using this metric, the operator can establish the *maximum* permissible percentage of exposed devices, according to the dimension of the network. For example, in the case of environments made by a consistent number of devices, the management becomes more complex, and it is almost impossible to handle all the devices exposed. In this case, the goal could be to reduce the value of *PoDD* whenever possible.

Chapter 5

Countermeasures

In this chapter, we describe the various strategies that the operator can put in place to strengthen the traffic of a vulnerable LoRa network. The goal is to reduce the amount of exposed ED recognized by PIVOT, modifying the devices' configuration to avoid being detected again by the system. From a technical point of view, the administrator should reduce the values of *NoDD* (Number of Detected Devices) and the *PoDD* (Percentage of Detected Devices) metrics. NoDD and PoDD are directly proportional, but they can refer to different contexts. Generally speaking, NoDD is designed to describe small networks, where devices have more predictable behavior. PoDD describes better those situations where the number of devices is much greater. For example, given a network with twelve vulnerable devices out of a total of twenty active ones, we have the following metrics $NoUD = 20$ and $NoDD = 12$, from which we derive $PoDD = \frac{NoDD}{NoUD} = 0.6$. The output of this metric could indicate a critical status of the network. However, if the operator compares the high value of PoDD to the lower value of NoDD, it recognizes that it has to modify the behavior of only twelve devices. It might be a more alarming circumstances if $PoDD = 0.6$ but $NoUD = 1000$!

PIVOT alters the operator when a device has a too predictable behavior that could expose it to unauthorized information disclosure, such as the association of the *DevAddr* identifiers to the global *DevEUI* address. The strategy behind the PIVOT detection algorithm is the *pattern-matching.* Therefore, the positivity of the outcome is closely related to the periodic pattern of the target device. This prerequisite implies that all countermeasures that do not alter the regularity in the ED flow prove to be ineffective. Our challenge is to change the pattern of the exposed device without compromising the purpose for which it is created. For example, if the target ED is an industrial pressure sensor designed to periodically send updates to the server, the operator should keep in mind that changing the behavior of the device could alter this basic functioning. Since we want to avoid that eavesdroppers can extrapolate the devices' patterns from analyzing the LoRa flow, the solutions we will describe are related to the family of the *pattern detection* countermeasures.

The first strategy we present is the addition of the so-called dummy packets to EDs transmissions. In the LoRaWAN frame, only the payload is encrypted, making these dummy messages indistinguishable from those containing the information. On the other side, the application server, provided with a packet filter, can distinguish real data from fake ones. Although several studies [26] show the effectiveness of this technique, we want to point out several disadvantages that lead us to exclude it as a final solution. First, this mechanism impacts performance, reducing channel availability. Moreover, the transmission of additional packets requires more energy than desired, inevitably reducing the lifetime of the ED. A implication that goes against the principles of LPWANs, such as LoRaWAN, designed with the principle to ensure the long duration of battery-operated devices. Finally, increasing the transition rate of the endpoints is a choice subject to various physical constraints. Indeed, as shown in figure 5.1, EDs can occupy the channel for a limited portion of time, called *duty cycle*. It is commonly set to 1%, or 1-time unit every 100-time units [2].

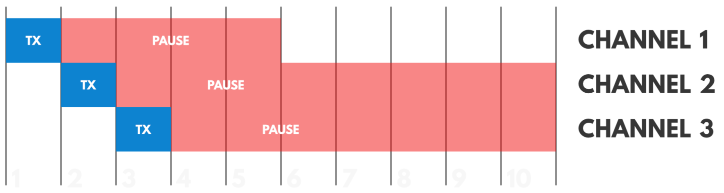


Figure 5.1. The duty cycle of LoRaWAN

In this work, we go deeper. As shown in figure 5.2, rather than altering traffic by adding new data to the network, we propose to hide the periodic behavior of devices by changing the flow generated by them. In this way, we don't add irrelevant data within the network, preserving the performance and costs required.

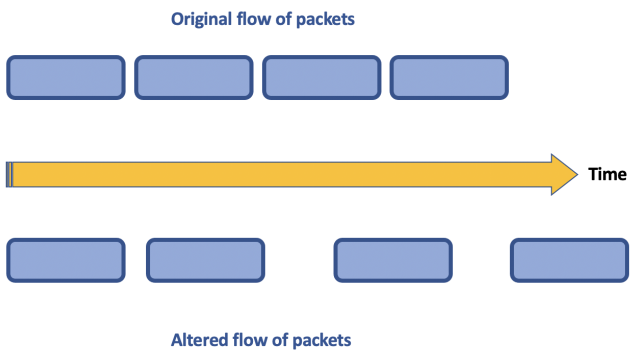


Figure 5.2. The altered flow

A basic approach is to regularize the flow by requiring endpoints to send packets following the same frequencies. Covering the different EDs' patterns, the behaviors of the various devices are indistinguishable. But it is not suitable for all types of applications. Indeed, many scenarios, such as alarming systems, require a fast response, and following a forced behavior than required could be a very restrictive limitation. Made these considerations, we propose to add a *random* delay before sending each packet. When the delay is high enough, the correlation between the device and its pattern is hidden. In detail, let be P the pattern as defined in Chapter 4:

$$P = \{s_1, ..., s_n\}$$

And the let be d_i the randomly generated delay to add to each segment s_i, then the we define the new pattern P' as:

$$P' = \{s_1 + d_1, ..., s_n + d_n\}$$

such that

$$P \not\equiv P'$$

for each d_i chosen.

The required delay can be extracted from a *pseudo-random* number sampling, a procedure for generating numbers distributed according to a given *probability distribution*. A well know pseudo-random number sampling method is the inverse transform sampling, which produces instances using the Cumulative Distribution Function (CDF) F_X of the random variable X. The function F_X takes as input some value x outputs the probability of obtaining $X \leq x$.

$$F_X(x) = P(X \leq x) = p$$

The inverse transform sampling uses the inverse of CFD. The F_X^{-1} takes p as input and returns x, where p is uniformly distributed. With this method, we can sample from any F_X if F_X^{-1} is known. We need just to take values $u \sim U(0,1)$ and then, through F_X^{-1}, obtain x's.

$$F_X^{-1}(u) = x$$

The exponential distribution

Our idea is to simulate a random variable X that follows the *exponential* distribution with mean λ i.e. $X \sim Exp(\lambda)$. In queueing theory, this distribution lends itself well to modeling customer interarrival times or service times for several reasons [6]. First, the exponential function is a strictly decreasing function of t, meaning that after arrival has occurred, the amount of waiting time until the next arrival is more likely to be small than large. Second, the exponential distribution is memoryless. This property implies that the time until the next arrival doesn't depend on the time that has already passed, preventing the adversary to extrapolate information about the behavior of the devices by analyzing their past transmissions. Specifically, if an adversary observes a packet after time t_1 and wants to calculate the probability of the next packet arriving after $t_2 = \Delta t + t_1$, the memorylessness property ensures that:

$$P(X > t_1 + \Delta t \,|\, X > t_1) = P(X > \Delta t)$$

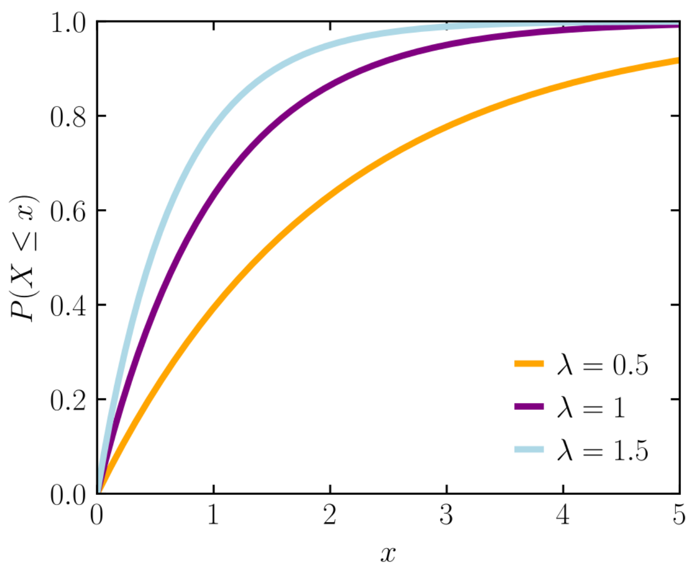


Figure 5.3. CDF of the exponential distribution

Going into the details, the Probability Density Function (PDF) of the exponential distribution is:

$$f(x,\lambda) = \lambda e^{-\lambda x}, x \geq 0, \lambda \geq 0$$

while the Cumulative Distribution Function (CDF), shown in Figure 5.3 is defined as follows:

$$F(x) = 1 - e^{-\lambda x}$$

By solving $p = F(x)$, we obtain the inverse function

$$x = F^{-1}(p) = -\frac{1}{\lambda}\ln(1-p)$$

The idea is illustrated in Figure 5.4. Random numbers are generated from a uniform distribution between 0 and 1. Each of the points is mapped according to $x = F_{-1}(y)$. We can observe that using this strategy, most of the points end up close to 0, while only a few of them end up having high values.

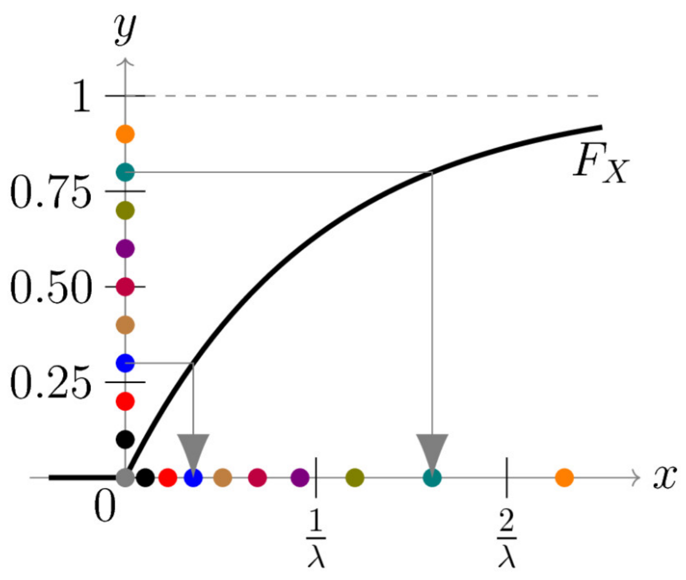


Figure 5.4. Inverse transformation method for exponential distribution

Chapter 6

Implementation and Results

In this chapter, we test our system and discuss the results. First, we illustrate how the simulation is carried out, and then, we explain our methodology to test PIVOT, defining the parameters and how they affect the simulations.

6.1 Implementation

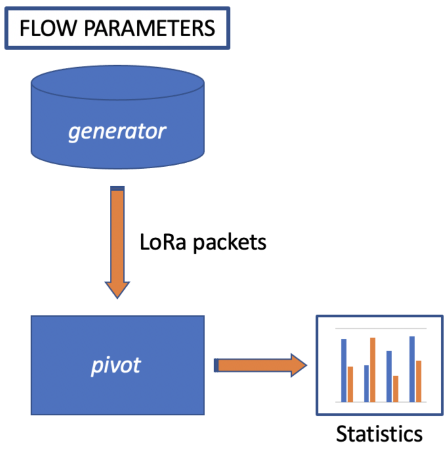


Figure 6.1. The architecture of the simulator

PIVOT is tested by running it in a simulated LoRaWAN network. As shown in figure 6.1, the simulator is made up of two components, the *generator* and *pivot*. The LoRa traffic is set and created by the *generator* to be subsequently read and analyzed by *pivot*. In detail, the *generator* takes as input a set of parameters that determine the characteristics of the flow, such as the number of devices generating packets or the transmission frequency. On the other side, *pivot* outputs the list of statistics metrics, described in Chapter 4. The simulation program was created using Python 3.7 and its source code is fully observable and downloadable from Github [29].

6.1.1 Code description

In this section we analyze in detail the *generator* and *pivot* components, displaying code snippets where relevant.

generator

The generator subroutine outputs a synthetic LoRa dataset, whose components are instances of the `Packet` class, shown in the following snippet.

```python
class Packet:
        def __init__(self, t, dev, devaddr, rssis, uid, fcnt, mtype,
        ↪  info=""):
                self.rssis = rssis  #can be None, if unavailable
                self.dev_eui = dev
                self.dev_addr = devaddr
                self.t = t
                self.uid = uid
                self.fcnt = fcnt
                self.mtype = mtype
                self.info = info
```

The class variables are well-known LoRa header fields, such as the two identifiers, the DevAddr and the DevEUI, the Message type *mtype* and the timestamp *t*.
In the original LoRaWAN specification there are six different message types, differently codified in the *mtype* fields. Since PIVOT filters only *uplink* and *Join-request* messages, the sole ones required, our simulator avoids generating unnecessary messages. The *mtype* of packets in the dataset are set exclusively to `000` (Join-request) and `010` (Unconfirmed Data Up). Moreover, LoRaWAN requires encrypting the payload using AES. Since the covered information cannot be read by external parts, such as PIVOT, we decide to omit this field in the class.
The DevAddress and DevEUI parameters also have a slightly different syntax from the official guidelines. The value of the DevEUI is given by an index i used to represent the device i-th in the simulated network. The DevAddress is the concatenation of the DevEUI and the j index, a counter that indicates the number of Join-request messages sent by the i-th device.

These measures are conceived to forge packets containing only essential elements for our purposes. Smaller packets imply large datasets to use in our tests.

```
N = 300      # number of devices
S = 365*24*3600      # number of seconds to generate packets
Tmin = int(10)      # minimum interarrival time, in seconds
Tmax = int(23*3600)      # maximum interarrival time, in seconds
Emin = 0.01      # min absolute error in the interarrival time
Emax = 2         # max absolute error in the interarrival time
P = 10       # maximum length of a pattern
Jmin = 20        # minimum number of messages before a join
Jmax = 300       # maximum number of messages before a join
```

The code above reports the set of parameters used to determine the characteristics of the traffic flow. The first variable to note is N, the number of *active* devices in the network i.e. all that endpoints that from the moment PIVOT is turned on send at least one frame in the network. N implicitly also represents the maximum number of devices that our system can detect. In the best scenario, given the vector C used by PIVOT to store all the identified patterns, we have $length(C) = N$.
The parameter P represents the maximum length of the pattern i.e. the number of segments $s_1, ..., s_n$ that compose the chain while $Tmin$ and $Tmax$ are respectively the smallest and the largest values that the segment s can take.
Using with three keys, the generator elaborates a new random pattern in this way:

```
# generate random pattern for the current device
pattern_len = randint(1, P)
pattern = [randint(Tmin, Tmax) for _ in range(pattern_len)]
```

To make the fake traffic as likely as possible to the real one, the simulator inserts a small error during the creation of packets. The system doesn't rigidly generate new packets following the established pattern, but supposes that the time distance between two messages (i.e. a *segment*), could be longer or shorter than expected. Then, it randomly chooses an error E in the range $[Emin, Emax]$ alters slightly the segment in the following way:

```
# generate packets following the pattern up until S seconds
packets_dev = []
i_pattern = 0
t = first_packet_t
while t < S:
    packets_dev.append(next_packet)
    t_err = (1.0 if random()>0.5 else -1.0) * (random() * (Emax -
    ↪  Emin) + Emin)
    # It could be the case that t_err is negative and |t_err| >
    ↪  pattern[i_pattern].
    # In this case the following assert yields an error
    next_packet_t = t + pattern[i_pattern] + t_err
    if t_err < 0 and abs(t_err) > abs(pattern[i_pattern]):
        # next_packet_t is set to be 10 seconds after the previous
        ↪  one
        next_packet_t += abs(t_err) - abs(pattern[i_pattern]) + 10
    assert(next_packet_t > t)
    next_packet = Packet(next_packet_t, str(dev_i), "---", None, None,
    ↪  -1, "Unconfirmed Uplink")

    i_pattern = (i_pattern+1) % pattern_len
    t = next_packet_t
```

Finally, we introduce the last two parameters, *Jmin* and *Jmax*, the minimum and the maximum number of messages before a join. They are a keystone for the correct functioning of PIVOT as they influence the accuracy of the detection algorithm. Indeed, if a device sends many packets before sending a *Join-request*, the probability that the system identifies its pattern is very high.

```
# add join messages for the current device
joins_dev = []
i = randint(Jmin, Jmax)
while i < len(packets_dev)-1:
    t1 = packets_dev[i].t
    t2 = packets_dev[i+1].t
    assert(t2 > t1)
    join_msg_t = t1 + (random() * (t2-t1))
    assert(t1 < join_msg_t < t2)
```

```
       join_packet = Packet(join_msg_t, str(dev_i), "not_available",
       ↪  None, None, -1, "Join Request")
       joins_dev.append(join_packet)
       i += randint(Jmin, Jmax)
```

patterns

The following code snippet reports the constructor of the `Pattern` class. As reported in Chapter 4, PIVOT stores each detected pattern along with the last *timestamp*, the chain of *segments* and the $verified$ flag, initialized to False.

```
class Pattern:
    def __init__(self, timestamp):
        self.timestamp = timestamp
        self.verified = False
        self.segments = []
```

Each element in the chain of segments is an instance of the `Segment` class. In detail, a segment s is stored with a counter n and an value $mean$, two parameters used to reduce the percentage of errors in defining the exact length of the segment.

```
class Segment:
    def __init__(self, value, index):
        self.n = 1
        self.index = index
        self.mean = value
```

As we have seen previously, a device does not always strictly follow its pattern, sending sometimes packets a little earlier or later than required. Then, the $mean$ represents the average size, in seconds, of the segment s in a given pattern.

Going into details, for each new DevAddress a, PIVOT initialize an empty pattern P_a. Then, when it receives a new packet p with the DevAddress a, it triggers the *update* subroutine:

```
def update(self, timestamp):
    old_t = self.timestamp
    self.timestamp = timestamp
    x = self.timestamp - old_t

    found = False
    for s in self.segments:
        if abs(s.mean - x) < e:
            found = True
            s.update(x)

    if found:
        self.verified = True

    if not found:
        # new segment
        index = len(self.segments) - 1
        segment = Segment(x, index)
        self.segments.append(segment)
        self.verified = False
```

This function calculates the current segment s, checks if it is in the chain of the pattern, and, if needed, updates the flag *verified*. In observing that the segment already belongs to the chain, the algorithm considers a possible margin of error e. When the assertion is verified, PIVOT triggers the method *update* of the class `Segment`:

```
def update(self, value):
    self.n += 1
    old_m = self.mean
    self.mean = old_m + ((value - old_m) / self.n)
```

This procedure is used to level the value of the *mean* parameter so that, despite the errors in the transmission, it can remain as accurate as possible to the real length of the segment. The updated value is the outcome of Welford's online algorithm [30], which computes the mean in a single pass, inspecting each value only once.

The last method of the `Pattern` class we are going to describe is *equals*, which code is reported below:

```
def equals(self, pattern):

    if len(pattern.segments) != len(self.segments):
        return False

    segments = pattern.segments
    for s in segments:
        if not s.belongs_to(self):
            return False

    return True
```

This procedure verifies a matching between two patterns P_1 and P_2. They are equal when the two chains are of the same length, with the same number of segments of equal length, sequentially distributed in the same way. When all these requirements are confirmed, the function returns `True`.

pivot

The following snippet displays the constructor of the `PIVOT` class:

```
class PIVOT:
    def __init__(self):
        self.confirmed = {}
        self.unconfirmed = {}
        self.quarantine = {}

        self.to_analyze = {}
        self.current_section = 1
```

The *confirmed*, *unconfirmed*, and *quarantine* variables represent the C, U, and Q vectors described in Chapter 4. These Python dictionaries stores the couples <dev_addr, pattern>, where "dev_addr" is the well-known identifier of the device and "pattern" is an instance of the Pattern class associated with the DevAddress. In detail, *confirmed* contains all the unique patterns detected by PIVOT, *unconfirmed* collects all the undetected ones i.e. associated with devices not yet recognized and finally *quarantine* has all that patterns that matches with almost one of those that are in *confirmed*.
to_analyze, that represent the data structure TA, stores the pairs <dev_addr, list_to_analyze>, where the key "dev_addr" is an unknown DevAddress, which

pattern P is in *unconfirmed*, and "list_to_analyze" is a list of candidate patterns to compare with P.
Finally, *current_section* is a counter used to estimate the number of section i.e. the amount of intervals between two Join-requests.

After the initialization of PIVOT, to simulate an incoming data stream, the dataset outputted by the *generator* is processed through a for loop, within which the *read_packet* method is triggered.

```
# generating the dataset
generate_synt_traffic(400)

# loading the dataset
packets =  pickle.load(open("synth_traffic.pickle", "rb"))

# new instance of PIVOT
pivot = PIVOT()

# PIVOT on listening
# analyzing the dataset
for p in packets:
    pivot.read_packet(p)
```

The function *read_packet* processes the incoming packet, extracting and analyzing the `mtype` field. As long as this value is different from "Join Request" and the current section is 1, PIVOT triggers the internal *pre_join* subroutine. Otherwise, it calls the *main* one.

```
def read_packet(self, p):
    if (p.mtype == "Join Request"):
        self.current_section += 1

    else:
        if self.current_section == 1:
            self.__pre_join(p)
        else:
            self.__main(p)
```

The *pre_join* function is a subroutine of the detection algorithm. As explained in Chapter 4, in this step the first patterns are identified and modeled, to be subsequently used as a reference in the *main* procedure. When the current DevAddr is not a key of the *confirmed* dictionary, a new instance of the `Pattern` class is triggered. When instead it is a key, the associated pattern is extracted and updated triggering the *update* method of the `Packet` class.

```python
def __pre_join(self, p):
    devaddr = p.dev_addr

    if devaddr in self.confirmed:
        self.confirmed[devaddr].update(p.t)
    else:
        self.confirmed[devaddr] = Pattern(p.t)
```

When the function *read_packet* processes an uplink packet and the current section is greater than 1, it calls the internal *main* subroutine. This procedure analyzes the DevAddress and checks progressively if it a key of the *confirmed* dictionary or of the *unconfirmed* one.

```python
def __main_procedure(self, p):

    # gets the Dev Address
    devaddr = p.dev_addr

    # checks if the pattern already exists
    if devaddr in self.confirmed:
        # rest of code

    else:
        # check if the DevAddress has already been received
        if devaddr in self.unconfirmed:
            # rest of code

        else:
            # initializing a new pattern
            self.unconfirmed[devaddr] = Pattern(p.t)
            self.to_analyze[devaddr] = list(self.confirmed.keys())
```

If neither of these two conditions occurs, the DevAddress belongs to an unknown device. PIVOT creates a new instance of `Pattern`, inserts it in the *unconfirmed* dictionary, and sets the *to_analyze* list with all the potential patterns with which it must verify the match. Initially, this list is filled with all the keys of the *confirmed* dictionary.

Otherwise, if the DevAddress is a key of the *confirmed* dictionary, it belongs to a device that did not make any Join-request, and then, did not change its address. This scenario implies that the associated pattern should not used as comparison model for other patterns. The *clean* function is triggered.

```
if devaddr in self.confirmed:
    self.confirmed[devaddr].update(p.t)
    self.__clean(devaddr)
```

This method simply removes from all the *to_analyze* lists the value *devaddr* given as input.

```
def __clean(self, devaddr):
    to_analyze = self.to_analyze.copy()

    for elem in to_analyze:
        if devaddr in to_analyze[elem]:
            self.to_analyze[elem].remove(devaddr)
```

Now, we analyze the case where the DevAddress is a key of the *unconfirmed* dictionary. First of all, PIVOT check if this address belongs to the *quarantine* dictionary also and, eventually, triggers the quarantine subroutine. Otherwise, it starts the matching procedure. It takes all the addresses stored in *to_analyze[DevAddress]* and extracts their respective patterns from *confirmed*. To ensure greater accuracy, only those patterns whose *verified* flag is set to `True` are taken into account. This newly obtained list is analyzed through a for loop. For simpler notation, we denote the pattern of the unconfirmed DevAddr as `unconf_pattern` and the current verified pattern we are going to compare with as `conf_pattern`. When there is a matching between the two patterns, their respective DevAddress are inserted in the *quarantine* dictionary so that the next time a new packet arrives with that DevAddress, PIVOT will trigger the *quarantine* subroutine. In all other cases where there is no match, the *to_analyze* list is updated, removing from it the address associated to the current conf_pattern.

```
if devaddr in self.unconfirmed:
    self.unconfirmed[devaddr].update(p.t)

    if devaddr in self.quarantine:
        self.__quarantine(devaddr, p.t)

    else:
        to_analyze = self.to_analyze[devaddr]
        unconf_pattern = self.unconfirmed[devaddr]


        verified = list(filter(lambda x: self.confirmed[x].verified,
        ↪  to_analyze))

        for v in verified:
            conf_pattern = self.confirmed[v]

            if len(unconf_pattern.segments) ==
            ↪  len(conf_pattern.segments):
                pattern_matching =
                ↪  conf_pattern.contains(unconf_pattern)

                if pattern_matching:
                    self.quarantine[devaddr] = (v, p.t)
                else:
                    self.to_analyze[devaddr].remove(v)

        if len(self.to_analyze[devaddr]) == 0:
            self.__new_device(devaddr, unconf_pattern)
```

At the end of the loop, if the *to_analyze* list is empty, then `unconf_pattern` hasn't had any matches and necessarily it is linked to a new device. PIVOT calls the function *new_device*, that registers the `unconf_pattern` in the *confirmed* dictionary and updates all the data structures, removing the DevAddress from *unconfirmed* and deleting the *to_analyze* list.

```python
def __new_device(self, devaddr, unconf_pattern):
    self.confirmed[devaddr] = unconf_pattern
    self.unconfirmed.pop(devaddr)
    self.to_analyze.pop(devaddr)
```

Finally, we illustrate the *quarantine* subroutine, reported in the following snippet. This procedure takes as input the variable **`devaddr`** i.e. the unknown DevAddress of which we must identify the device, and triggers an alarm if it detects that **`devaddr`** belongs an already joined device. The *quarantine* makes use of two parameters, **`pattern`**, that represent the pattern that matches with the one associate to **`devaddr`**, and **`suspect`**, that is the DevAddress correlated to **`pattern`**.
This function first calculates the current segment associated to **`devaddr`**, then checks if it is in the chain of **`pattern`**. In that case, we have the confirm that the **`suspect`** and the **`devaddres`** belong to the same device. The alarm is triggered and the *confirmed* dictionary is updated, replacing the key **`suspect`** with **`devaddr`** since it is the current address of that device. Moreover, the function *clean* is called so that the variable **`suspect`** is removed from the *to_analyze* lists (therefore it will no longer be taken into consideration in the matching procedures).
On the other hand, if the segment don't belong to the chain of **`pattern`**, the variable **`devaddr`** represents a new device, and it is added as a new key in *confirmed.*

```python
def __quarantine(self, devaddr, p_timestamp):
    suspect = self.quarantine[devaddr][0]
    timestamp = self.quarantine[devaddr][1]
    pattern = self.confirmed[suspect]

    x = Segment(p_timestamp - timestamp, 0)
    if x.belongs_to(pattern):
        self.confirmed[devaddr] = self.unconfirmed[devaddr]
        self.confirmed.pop(suspect)
        self.__clean(suspect)

    else:
        new_pattern = self.unconfirmed[devaddr][suspect]
        self.confirmed[devaddr] = new_pattern

    self.unconfirmed.pop(devaddr)
    self.quarantine.pop(devaddr)
    self.to_analyze.pop(devaddr)
```

At the end of *quarantine*, since PIVOT has successfully identified the nature of `devaddr`, it removes it from *unconfirmed*, *quarantine* and *to_analyze* structures.

metrics for the operator

The following snippet reports the four metrics that PIVOT outputs to the operator.

```
def metrics(self):
    number_of_unique_devices = len(self.confirmed)
    number_of_detected_devices = self.detected
    return {
        "NoUD" : len(self.confirmed),
        "NoJ" : self.current_section - 1,
        "NoDD": self.detected,
        "PoDD" : number_of_detected_devices /
        ↪  number_of_unique_devices
    }
```

6.2 Results

In this section, we show the results of the various tests carried out using the simulator. In particular, we first analyze the *sensitivity* of PIVOT, showing which parameters can affect the correct functioning of the system. Subsequently, we will compare two different scenarios to demonstrate how by applying countermeasures described in Chapter 4 the operator can effectively reduce the vulnerability of exposed devices. All the evaluations are run with Python3.7 on a x64 laptop equipped with a Intel Core i7 CPU @2.50GHz and 8 GB of RAM.

6.2.1 Sensitivity of the system

Basic performance measures can be derived from the *confusion* matrix, shown in figure 6.2. It is a two-by-two table containing four outcomes produced by a binary classifier, used to derive well-known statistical criteria, such as error rate, accuracy, specificity, sensitivity, and precision.

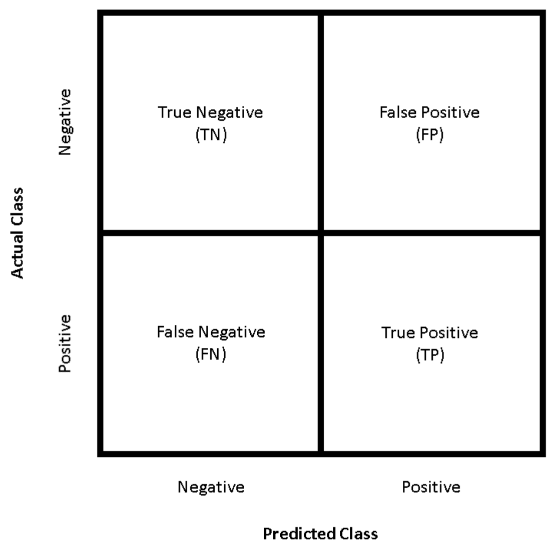


Figure 6.2. Confusion matrix structure for binary classification problems

For our tests, we define the *positive* and *negative* classes in this way:

- The **positive** class represents the number of times in which devices, sending a

Join-request, change their DevAddress. This value can also be interpreted as the total number of times PIVOT could potentially recognize a match between two patterns.

- The **negative** class is the number of ED that have never changed their address. PIVOT should be able to classify them as *new* devices.

Going into detail, in a binary classification like the one just described, there are four possible outcomes.

- `True positive (TP)`: correct positive prediction
- `False positive (FP)`: incorrect positive prediction
- `True negative (TN)`: correct negative prediction
- `False negative (FN)`: incorrect negative prediction

According to the positive and negative classes defined above, these four metrics can be interpreted in the following way:

- `True positive (TP)`: PIVOT recognizes a match between two patterns and they belongs to the same ED.
- `False positive (FP)`: PIVOT recognizes a match between two patterns but they belongs to different EDs.
- `True negative (TN)`: PIVOT correctly classifies a DevAddr as belonging to a new device.
- `False negative (FN)` : PIVOT classifies a DevAddr as belonging to a new device, but it is an already joined device that changes its address.

From these values, we can compute the `True Positive Rate (TPR)`, commonly known as sensitivity or recall. The technical definition of TPR is the number of true positives divided by the number of true positives plus the number of false negatives.

$$TRP = \frac{TP}{TP + FN}$$

In other words, it represents the ability our model to find all the relevant matches within the data set and, if compared to other more well-known metrics, such as accuracy, it allows us to draw more specific conclusions from the various tests. Indeed, despite accuracy being a more intuitive performance measure, it is simply a ratio of correctly predicted observations to the total observations and works better with symmetric datasets where the number of false positive and false negatives are almost the same. Imbalanced classification tasks requires more sophisticated statistical parameters such as precision and, precisely, recall, from the combination of which it is possible to obtain useful models, such as the F1 score.

Table 6.1. The two variables used to analyze the sensitivity of the detection algorithm.

Parameter	**Description**
`N`	Number of active devices in the network
`P`	Maximum length of patterns

Given the definition of recall, we now illustrate how the different parameters can influence the value of this metric. Table 6.1 reports the two variables we choose to determining the characteristics of the traffic generated by the simulator. In the following tests, we use them to analyze the sensitivity of the PIVOT detection algorithm by varying a chosen variable and fixing the remaining one to predetermined values.

Parameter N

The first variable in analysis is N, the overall number of *active* devices in the network i.e. all that ED that sends almost one packet while PIVOT is running. In detail we conducted 31 different measurements, ranging the value of N from 50 to 200, with an increment of 5 after each iteration. The figure 6.3 reports the results of the evaluation:

Table 6.2. The value of the parameters used to generate the traffic flow for this test.

Parameter	**Value**
N	Variable
P	5
J	[20, 300]
E	[0.01, 2]

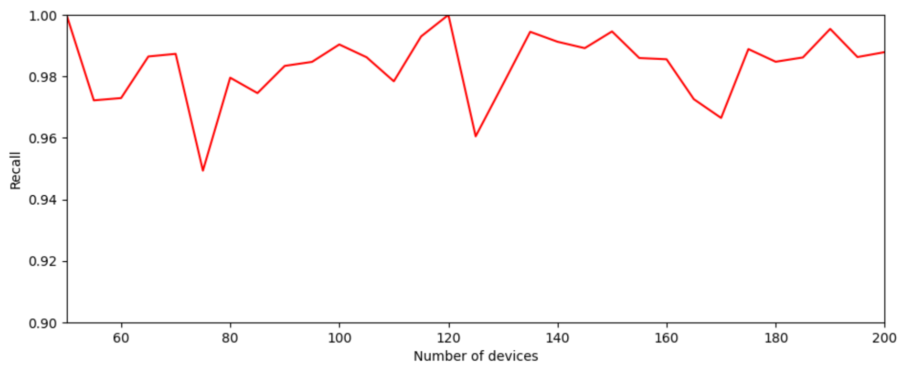


Figure 6.3. This figure shows the recall value as parameter N changes. The sensitivity of the system scales from from the lowest 0.94 to the highest 1.0, with an average value of 0.98

The recall ranges from the lowest 0.94 to the highest 1.0, with an average value of 0.98. This small range of values demonstrates that PIVOT identifies almost all vulnerable devices regardless of the number of endpoints in the network.

Parameter P

Now we analyze how recall varies according to parameter P, the maximum length that a pattern can have within the network. We tested 29 different scenarios, with a value of P ranging from a minimum of 2 to a maximum of 30.

Table 6.3. The value of the parameters used to generate the traffic flow for this test.

Parameter	**Value**
N	100
P	Variable
J	[20, 300]
E	[0.01, 2]

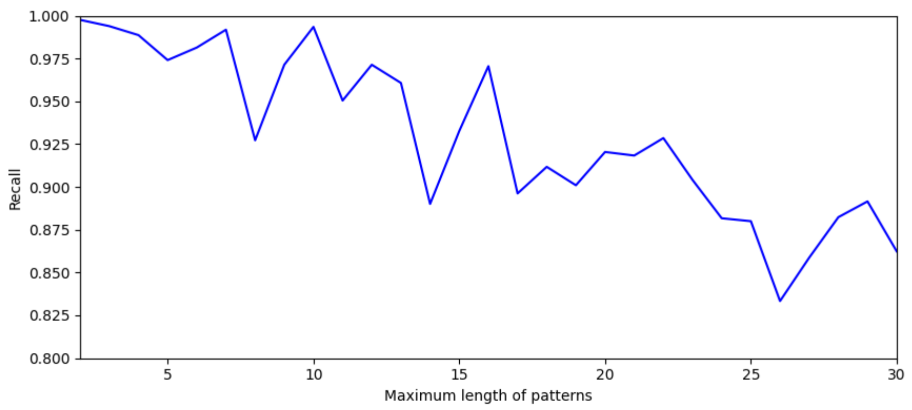


Figure 6.4. This figure shows the recall value as parameter P changes. The sensitivity of the system scales from a minimum of 0.83 to a maximum of 0.99, with a mean of 0.92

In this case, the results scales from a minimum of 0.83 to a maximum of 0.99, with a mean of 0.92. However, as shown in figure 6.4, the detection algorithm is less effective as the length of the pattern increases. Indeed, PIVOT takes longer to recognize complex patterns, consequently leaving many addresses waiting to be classified. In any case, the effectiveness of the algorithm does not experience strong changes in the presence of shorter patterns. And since most LoRaWAN devices have just this kind of patterns, this represents an advantage.

6.2.2 Testing the countermeasures

In this section we show that the strategies explained in Chapter 4 can effectively increase the security level of a LoRaWAN network. In detail, our demonstration consists of two steps. First, we generate and analyze a normal LoRa traffic flow then, we alter the inter-arrival time between packets by adding an *exponential delay*, and, finally, we use the metrics produced by PIVOT to compare the two scenarios.

The following code snippet reports the *add_exp_delay* function, used by the generator to change the traffic flow (represented by the dataset `patterns`) given in input.

```
def add_exp_delay(packets, exp_rate):
    assert(exp_rate>0)

    # total delay of all the packets, in seconds
    exp_total_delay = 0
    # list percentual increases of the delay
    exp_delay_inc = []
    # number of exponential arrival not used. An exponential arrival
    ↪  is not used if it arrives before the original interarrival
    ↪  time of the packet
    exp_total_notused = 0

    devs = set([p.dev_eui for p in packets])  #set of all devices

    for dev in devs:
        packets_dev = [p for p in packets if p.dev_eui == dev]

        # init exponential interarrival times generator
        exp_gen = exp_generator(exp_rate)

        # modify packet arrival times
        t_exp = packets_dev[0].t
        prev_t = t_exp
        for p in packets_dev[1:]:
            t_exp += next(exp_gen)
            while p.t > t_exp:
                # exponential arrival not used, it is before the
                ↪  original interarrival
                exp_total_notused += 1
                t_exp += next(exp_gen)
            # update stat
            exp_total_delay += t_exp - p.t
            assert(p.t - prev_t >= 0)
            exp_delay_inc.append((t_exp - p.t)/(p.t - prev_t))
            ↪  #ratio of the delay over the original interarrival
            ↪  time
            # modify packet time
            prev_t = p.t
            p.t = t_exp

    return packets
```

In detail, a for loop processes each packet p, modifying the associate timestamp by adding a randomly generated delay, generating using the *exp_generator* function.

```
def exp_generator(exp_rate):
    while True:
        yield from np.random.exponential(1/exp_rate, size=(10**5,))
```

This method uses the `np.random.exponential` function of *NumPy*, the fundamental package for scientific computing in Python. It takes as input the so-called scale rate, or `exp_rate`, a parameter used to determine the *spread* of the distribution. Due to this characteristics, it plays a fundamental role in determining the "weight" that the exponential delay assumes within traffic flow. Lower `exp_rate` values define greater delays.

We performed four different evaluations. The traffic is generated using the following universal parameters:

Table 6.4. The value of the parameters used to generate the traffic flow for the four tests.

Parameter	**Value**
N	100
P	5
J	[20, 300]
E	[0.01, 2]

Table 6.5. The table reports the results of four different measurement. In the first one no delay is applied to the inter-arrival times of the packets. It can be seen that by applying the countermeasures the detection is considerably reduced

	Test 1	**Test 2**	**Test 3**	**Test 4**
Exp. Rate	*None*	*0.1*	*0.05*	*0.01*
NoDD	136	3	2	0
NoUD	100	101	101	100
PoDD	1.36	0.029	0.019	0.0

Table 6.5 reports the results of our simulations, where *NoDD*, *NoUD* and *PoDD* correspond to the metrics outputted by PIVOT at the end of each test. As it is clearly visible, the values of NoDD and PoDD decrease considerably as the scale parameters vary, until they assumes a null value in the last test. In detail, from Test 1 and Test 2, where we apply the exponential delay for the first time, *NoDD* passes from 136 to 3, with an incredible reduction of 97%. Not only, from Test 2 to Test 4, where we reduce the value of the *exp_rate* to increment the delay, this percentage goes to 100%.

Chapter 7

Conclusions

In this thesis, we have examined that although the LoRaWAN protocol is designed to be secure and reliable, it suffers from still unsolved privacy issues, which an unauthorized eavesdropper can exploit to obtain unauthorized information about devices and users of a LoRa network. In particular, by investigating the different existing attack techniques, we have concluded that by analyzing the behavior of the devices in the traffic, an attacker, contrary to what is stated in the LoRaWAN specification, can correctly identify with high accuracy the couples <DevAddr, DevEUI> of the endpoints. To preserve the confidentiality of the data transmitted even in presence of these traffic analysis techniques, we presented a preventive strategy named PIVOT, a privacy monitoring system to integrate into an exposed network, which, by analyzing the traffic generated by the endpoints, identify possible critical situations and possibly alert the network operator. In detail, PIVOT takes as input the LoRa flow, elaborates it in real-time through a *detection* algorithm, and outputs different metrics about the status of the network. In the last part of the work, we introduced a Python simulator to test PIVOT and discussed the results obtained. In the evaluation procedure, we first examined the *sensibility* of the system i.e. the ability to correctly recognize all possible matches when some variables of the flow change. Then, in a second test, comparing two LoRa flows, we demonstrate that if the operator alters the transmission frequencies of the EDs by adding delays between the inter-arrival times of packets, the percentage of detections by PIVOT drops to 97%, effectively making the network less vulnerable.

The detection algorithm of PIVOT proves to be very accurate and able to recognize a high percentage of vulnerable devices, regardless of the number of active EDs in the network. On the contrary, it suffers slightly when the patterns to be identified become more complex. But it is an uncommon scenario in real LoRa networks, which by their nature, devices tend to transmit a small amount of data to guarantee maximum energy efficiency.

7.1 Next steps

PIVOT is only a starting point and offers significant opportunities for improvement. The detection algorithm, which is the heart of this system, performs a too strict pattern analysis, making the system not yet fully adaptable to more complex situations than those analyzed in this work, where devices may have unexpected transmission errors, sending for example abnormal packets, or may not necessarily follow any pattern. Furthermore, it uses a binary classification system that catalogs each DevAddress as belonging to a new device or a re-joined one. This scheme can be further refined by adding different levels of classification and alerting the operator only when the most critical one is reached. Finally, the *update* subroutine, which systematically analyzes the patterns and verifies their completeness, can be improved. Indeed, the `verified` flag, which is currently a Boolean variable, can be replaced by a numeric value in the range [0,1], where the extremes 0 and 1 respectively indicate the minimum and maximum probability that the detected pattern conforms to the original one, allowing greater accuracy in filtering reliable patterns to use during the matching procedure. In this way, the value of the flag does not change unexpectedly in the presence of packets whose transmission was not expected but gradually decreases. Future researches of this work will focus on refining the detection algorithm, possibly integrating it with stream-processing frameworks to speed up some steps and make detection even faster, regardless of the number of devices or packets transmitted. Finally, the PIVOT system can be expanded by adding new features, and modules or adapting it for other LPWAN technologies.

Bibliography

[1] Adefemi Alimi, K. O., Ouahada, K., Abu-Mahfouz, A. M., and Rimer, S. A survey on the security of low power wide area networks: Threats, challenges, and potential solutions. *Sensors (Basel, Switzerland)*, **20** (2020), 5800. 33066336[pmid]. Available from: `https://doi.org/10.3390/s20205800`, `doi:10.3390/s20205800`.

[2] Adelantado, F., Vilajosana, X., Tuset-Peiro, P., Martinez, B., Melia-Segui, J., and Watteyne, T. Understanding the limits of lorawan. *IEEE Communications Magazine*, **55** (2017), 34. `doi:10.1109/MCOM.2017.1600613`.

[3] Al-Hadhrami, Y. and Hussain, F. K. Ddos attacks in iot networks: a comprehensive systematic literature review. *World Wide Web*, **24** (2021), 971. Available from: `https://doi.org/10.1007/s11280-020-00855-2`, `doi:10.1007/s11280-020-00855-2`.

[4] Ancian, L. T. and Cunche, M. Re-identifying addresses in LoRaWAN networks. Research Report RR-9361, Inria Rhône-Alpes ; INSA de Lyon (2020). Available from: `https://hal.inria.fr/hal-02926894`.

[5] Barriga, J. J., Sulca, J., León, J., Ulloa, A., Portero, D., García, J., and Yoo, S. G. A smart parking solution architecture based on lorawan and kubernetes. *Applied Sciences*, **10** (2020). Available from: `https://www.mdpi.com/2076-3417/10/13/4674`, `doi:10.3390/app10134674`.

[6] Berry, R. Queuing theory (2006). Available from: `https://www.whitman.edu/documents/Academics/Mathematics/berryrm.pdf`.

[7] Brik, V., Banerjee, S., Gruteser, M., and Oh, S. Wireless device identification with radiometric signatures. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, MobiCom '08, p. 116–127. Association for Computing Machinery, New York, NY, USA (2008).

ISBN 9781605580968. Available from: `https://doi.org/10.1145/1409944.1409959`, `doi:10.1145/1409944.1409959`.

[8] Centenaro, M., Vangelista, L., Zanella, A., and Zorzi, M. Long-range communications in unlicensed bands: the rising stars in the iot and smart city scenarios. *IEEE Wireless Communications*, **23** (2016), 60. `doi:10.1109/MWC.2016.7721743`.

[9] Chacko, S. and Job, M. D. Security mechanisms and vulnerabilities in LPWAN. *IOP Conference Series: Materials Science and Engineering*, **396** (2018), 012027. Available from: `https://doi.org/10.1088/1757-899x/396/1/012027`, `doi:10.1088/1757-899x/396/1/012027`.

[10] Lorawan coverage (2021). Available from: `https://lora-alliance.org/`.

[11] de Moraes, P. and Conceição, A. F. D. A systematic review of security in the lorawan network protocol. *ArXiv*, **abs/2105.00384** (2021).

[12] El Fehri, C., Baccour, N., Berthou, P., and Kammoun, I. Experimental analysis of the over-the-air activation procedure in lorawan. In *2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 30–35 (2021). `doi:10.1109/WiMob52687.2021.9606301`.

[13] Farej, Z. and Maher, A. Performance comparison among (star, tree and mesh) topologies for large scale wsn based ieee 802.15.4 standard. *International Journal of Computer Applications*, **124** (2015), 41. `doi:10.5120/ijca2015905515`.

[14] Gu, X., Yang, M., Zhang, Y., Pan, P., and Ling, Z. Fingerprinting network entities based on traffic analysis in high-speed network environment. *Security and Communication Networks*, **2018** (2018), 6124160. Available from: `https://doi.org/10.1155/2018/6124160`, `doi:10.1155/2018/6124160`.

[15] Hupperich, T., Maiorca, D., Kührer, M., Holz, T., and Giacinto, G. On the robustness of mobile device fingerprinting: Can mobile users escape modern web-tracking mechanisms? In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, p. 191–200. Association for Computing Machinery, New York, NY, USA (2015). ISBN 9781450336826. Available from: `https://doi.org/10.1145/2818000.2818032`, `doi:10.1145/2818000.2818032`.

[16] Lauridsen, M., Vejlgaard, B., Kovacs, I. Z., Nguyen, H., and Mogensen, P. Interference measurements in the european 868 mhz ism band with focus on lora and sigfox. In *2017 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6 (2017). `doi:10.1109/WCNC.2017.7925650`.

[17] Leu, P., Puddu, I., Ranganathan, A., and Capkun, S. I send, therefore I leak: Information leakage in low-power wide area networks. *CoRR*, **abs/1911.10637** (2019). Available from: `http://arxiv.org/abs/1911.10637`, `arXiv:1911.10637`.

[18] Lorawan security - withepaper. Available from: `https://lora-alliance.org/wp-content/uploads/2020/11/lorawan_security_whitepaper.pdf`.

[19] Mekki, K., Bajic, E., Chaxel, F., and Meyer, F. A comparative study of lpwan technologies for large-scale iot deployment. *ICT Express*, **5** (2019), 1. Available from: `https://www.sciencedirect.com/science/article/pii/S2405959517302953`, `doi:https://doi.org/10.1016/j.icte.2017.12.005`.

[20] Lora message types (2021). Available from: `https://www.thethingsnetwork.org/docs/lorawan/message-types/`.

[21] Noreen, U., Bounceur, A., and Clavier, L. A study of lora low power and wide area network technology. In *2017 International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, pp. 1–6 (2017). `doi:10.1109/ATSIP.2017.8075570`.

[22] Oniga, B., Dadarlat, V., De Poorter, E., and Munteanu, A. Analysis, design and implementation of secure lorawan sensor networks. In *2017 13th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pp. 421–428 (2017). `doi:10.1109/ICCP.2017.8117042`.

[23] Philip, S. J., McQuillan, J. M., and Adegbite, O. Lorawan v1.1 security: Are we in the clear yet? In *2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys)*, pp. 112–118 (2020). `doi:10.1109/DependSys51298.2020.00025`.

[24] What are lora® and lorawan®? Available from: `https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan/`.

[25] Robyns, P., Marin, E., Lamotte, W., Quax, P., Singelée, D., and Preneel, B. Physical-layer fingerprinting of lora devices using supervised and zero-shot learning. In *Proceedings of the 10th ACM Conference on Security*

and Privacy in Wireless and Mobile Networks, WiSec '17, p. 58–63. Association for Computing Machinery, New York, NY, USA (2017). ISBN 9781450350846. Available from: `https://doi.org/10.1145/3098243.3098267`, `doi:10.1145/3098243.3098267`.

[26] Shbair, W. M., Bashandy, A. R., and Shaheen, S. I. A new security mechanism to perform traffic anonymity with dummy traffic synthesis. In *2009 International Conference on Computational Science and Engineering*, vol. 1, pp. 405–411 (2009). `doi:10.1109/CSE.2009.53`.

[27] Singh, R. K., Puluckul, P. P., Berkvens, R., and Weyn, M. Energy consumption analysis of lpwan technologies and lifetime estimation for iot application. *Sensors*, **20** (2020). Available from: `https://www.mdpi.com/1424-8220/20/17/4794`, `doi:10.3390/s20174794`.

[28] Spadaccino, P., Garlisi, D., Cuomo, F., Pillon, G., and Pisani, P. Discovery privacy threats via device de-anonymization in lorawan. In *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*, pp. 1–8 (2021). `doi:10.1109/MedComNet52149.2021.9501247`.

[29] Terenzi, F. Pivot source code. Available from: `https://github.com/francescoterenzi/PIVOT`.

[30] Welford, B. P. Note on a method for calculating corrected sums of squares and products. *Technometrics*, **4** (1962), 419. Available from: `https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022`, `arXiv:https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022`, `doi:10.1080/00401706.1962.10490022`.