GENERAL INTRODUCTION

The Internet of Things (IoT) has revolutionized the way we interact with technology, embedding connectivity into everyday objects from smart home devices to industrial sensors. What began in the early 21st century as a conceptual vision has matured into a vast ecosystem of interconnected systems, enabling seamless data exchange and automation across diverse industries. A key driver of this evolution is Low Power Wide Area Networks (LPWAN), a family of wireless communication technologies specifically designed to provide long-range connectivity for IoT devices while minimizing energy consumption.

Among LPWAN solutions, LoRaWAN (Long Range Wide Area Network) has emerged as a leading protocol, offering an efficient, scalable, and cost-effective way to deploy IoT networks. Its ability to transmit data over several kilometers while maintaining low power consumption. However, as LoRaWAN adoption grows, so does its exposure to cyber threats.

Unlike conventional IT systems, IoT deployments often operate in distributed, resource-constrained environments where even a single vulnerability can trigger widespread systemic failures. A breach in a LoRaWAN network does not merely risk data exposure; it can disrupt critical infrastructure, facilitate large-scale device hijacking, and pose serious physical safety risks particularly in sectors such as healthcare, smart cities, and industrial automation.

Although LoRaWAN incorporates essential security mechanisms such as encryption and mutual authentication at both the network and application layers certain threats persist. Device impersonation, key compromise, physical tampering, and side channel attacks remain serious threats. These ongoing vulnerabilities underscore the urgent need for robust, context-aware security measures.

In response to these challenges, our Final Year Project at the Higher School of Communication of Tunis SUP'COM is dedicated to a thorough exploration of the security landscape of LoRaWAN

networks. Our objectives include identifying and analyzing potential vulnerabilities, conducting attack simulations to assess real-world threats, and evaluating their impact on network integrity. Building on these insights, we aim to propose robust security measures that effectively mitigate identified risks and enhance the resilience and reliability of LoRaWAN-based IoT infrastructures.

Our report is structured into four comprehensive chapters:

- The first chapter presents the technical foundations of LoRaWAN technology, including its architecture, communication protocols, and security mechanisms. Details the project context, our objectives, and the methodology adopted for the study.
- The second chapter conducts an in-depth analysis of LoRaWAN security risks, focusing on critical vulnerabilities . Systematically categorizes attacks and their impacts on integrity, authentication, availability, and confidentiality.
- The third chapter delves into the design and implementation of attack simulations on LoRaWAN networks. This chapter discusses the complete setup of our LoRaWAN test environment, including the hardware and software stack.
- Finally, the fourth chapter demonstrates a practical attack chain from traffic interception to desynchronization attacks. Wraps up by proposing concrete security recommendations and mitigations strategies.

The report concludes with a general conclusion and perspectives for future developments.

Chapitre

1 Foundational Overview

Contents

Introduction

In this chapter, we present the context and motivation behind our project, followed by the technical foundations necessary to understand LoRaWAN networks. We then define the problem addressed, outline our objectives, and describe the methodology adopted for this study.

1.1 General context of the project

With the ongoing evolution of the Internet of Things (IoT) and the increasing adoption of LoRaWAN networks for low-power, long-range communication, the need to strengthen their security has become more critical than ever. These technologies are now deeply embedded in essential systems across various sectors, making them attractive targets for malicious actors.

It is within the framework of our Final Year Project at the Higher School of Communication of Tunis (SUP'COM) that we are carrying out an in-depth study on the security of LoRaWAN networks. Our work aims to identify potential vulnerabilities, simulate realistic cyberattacks, and propose effective security measures to enhance network resilience in real-world deployments.

1.2 Technical Background

1.2.1 LoRaWAN Overview

Internet of Things and the Rise of LPWAN

The Internet of Things is reshaping our interaction with the physical world by interconnecting billions of devices capable of sensing, communicating, and acting autonomously. From smart thermostats and wearable health monitors to connected cars and intelligent urban infrastructure, IoT spans a vast range of applications that enhance efficiency, safety, and user experience. This diversity has driven the development of communication protocols, each tailored to ensure reliability, scalability, and adaptability across various environments [1].

Conventional wireless networks like Wi-Fi or cellular are often unsuitable for constrained IoT nodes due to their power and range limitations. To address this, Low Power Wide Area

Networks (LPWANs) have emerged, supporting energy-efficient long-range communication at low cost. Technologies such as **SigFox**[1], **Weightless**[2], and **Ingenu**[3] compete in this space, but **LoRa** stands out for its balance of performance, affordability, and ecosystem maturity [2].

LoRa: The Physical Layer

LoRa (Long Range) is the modulation scheme underlying LoRaWAN. It employs chirp spread spectrum (CSS) modulation, offering resilience to interference, long-range capabilities, and low energy usage. A key parameter is the spreading factor (SF), which adjusts the trade-off between range and data rate. Higher SFs increase transmission time and range but reduce data rate, as illustrated in Figure 1.1 [3].

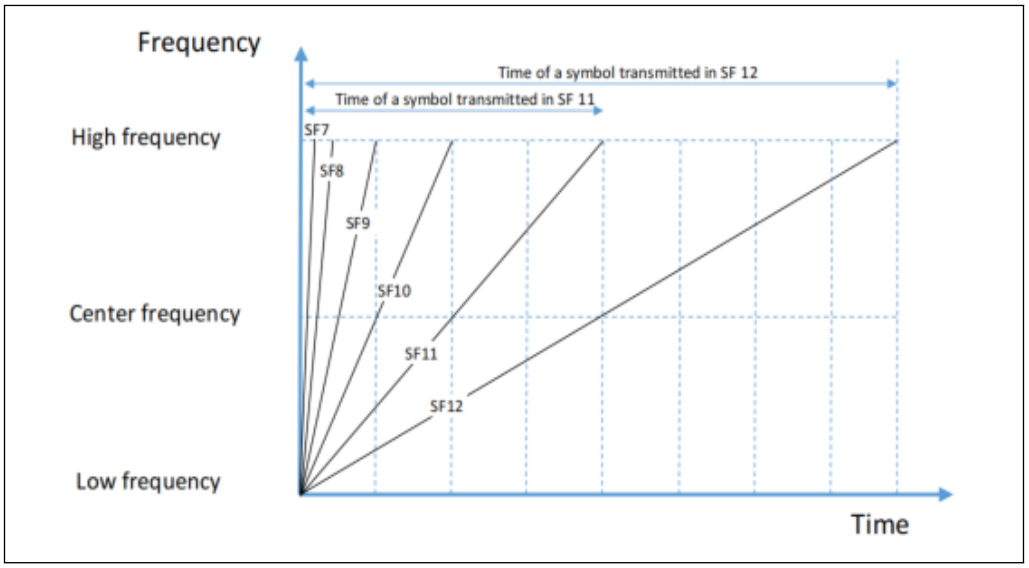


Figure 1.1: Symbol transmission time [4]

This modulation flexibility allows concurrent transmissions using different SFs, improving scalability and robustness [4].

LoRaWAN: The Protocol Layer

LoRaWAN builds upon LoRa by defining the network architecture, communication rules, and security mechanisms. Managed by the LoRa Alliance, the protocol is designed to support low-

[1]Long-range, low-energy technology for low-data-rate applications.
[2]LPWAN protocol with bidirectional communication and dynamic spectrum allocation.
[3]Uses RPMA (Random Phase Multiple Access) for robust, long-range IoT connectivity.

cost, interoperable, and secure communication. While LoRa focuses on how data is transmitted (modulation), LoRaWAN specifies how devices communicate and secure data (network protocol).

As represented in Figure 1.2, LoRaWAN has evolved through multiple versions since its inception in 2015. This work is based on version 1.0.3 for implementation while also considering version 1.1 for its improvements, as it is the most advanced version in terms of security features.

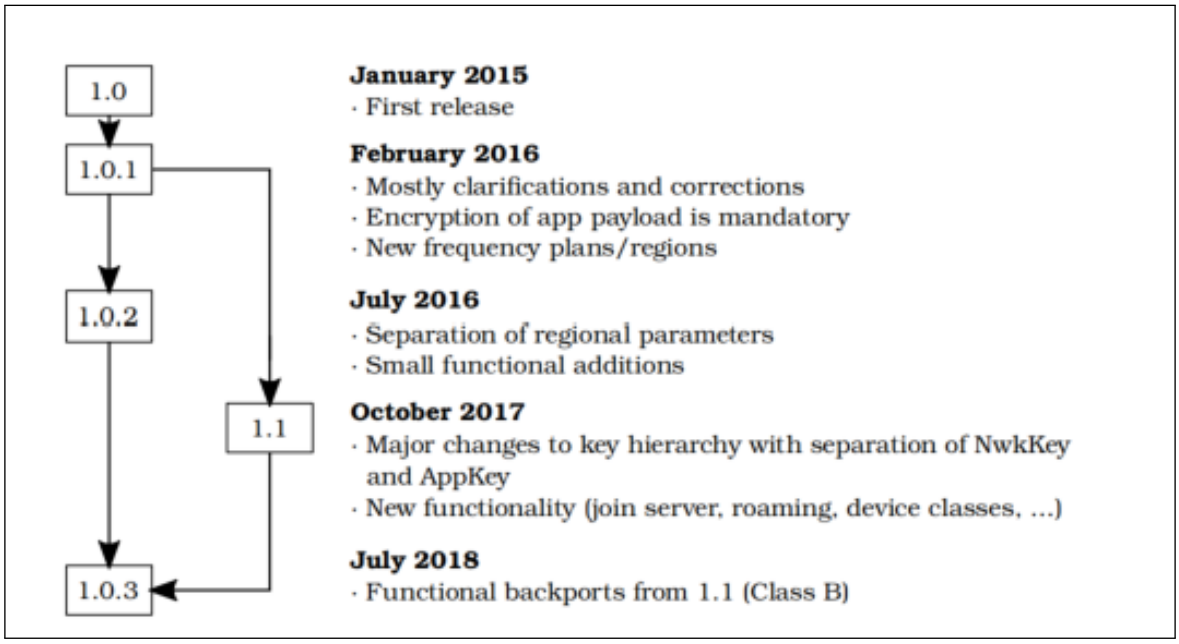


Figure 1.2: Version history of LoRaWAN [5]

LoRaWAN: Architecture

LoRaWAN adopts a star-of-stars topology: low-power end devices communicate wirelessly with multiple gateways via LoRa. These gateways act as transparent bridges, forwarding data packets to a centralized network server over standard IP connections as shown in Figure 1.3.

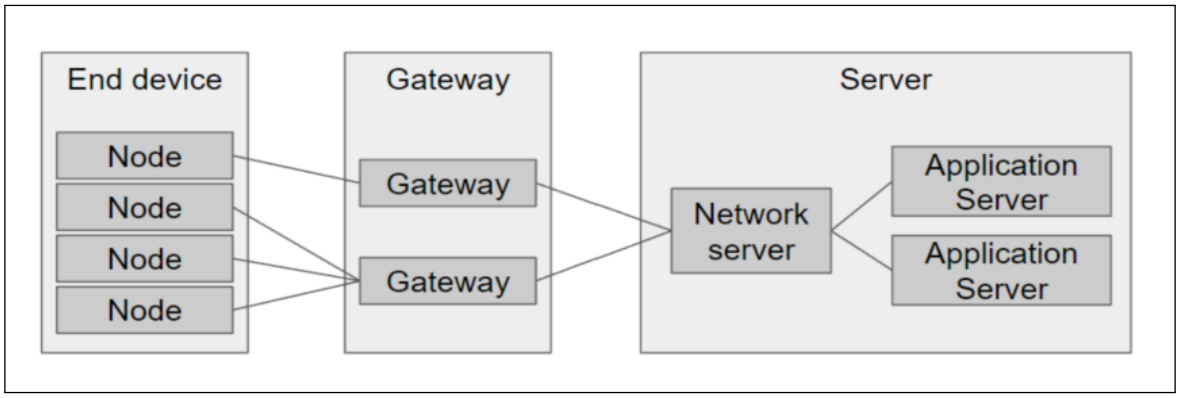


Figure 1.3: Architecture of a LoRaWAN network [1]

Unlike traditional cellular networks,where devices are bound to a specific base station, in LoRaWAN, any gateway that receives an uplink from an end device forwards it to the server, enhancing network reliability and redundancy [6].

End-Devices and Device Classes

LoRaWAN end-devices are compact, energy-efficient, and cost-effective IoT nodes designed for long-range communication. Equipped with LoRa transceivers, these devices can transmit data to any available gateway without a fixed link, ensuring scalable and reliable message delivery. These characteristics make them ideal for applications like environmental sensing, smart metering and asset tracking [4].

To optimize the balance between energy consumption and communication responsiveness, LoRaWAN defines three distinct operating modes, also referred to as device classes:

- **Class A – Ultra-Low Power:** This is the default mode supported by all LoRaWAN end-devices. After each uplink, two downlink reception windows (RX1, RX2) open. Outside these windows, the device remains in a deep sleep state, making this the most energy-efficient mode as demonstrated in Figure 1.4.

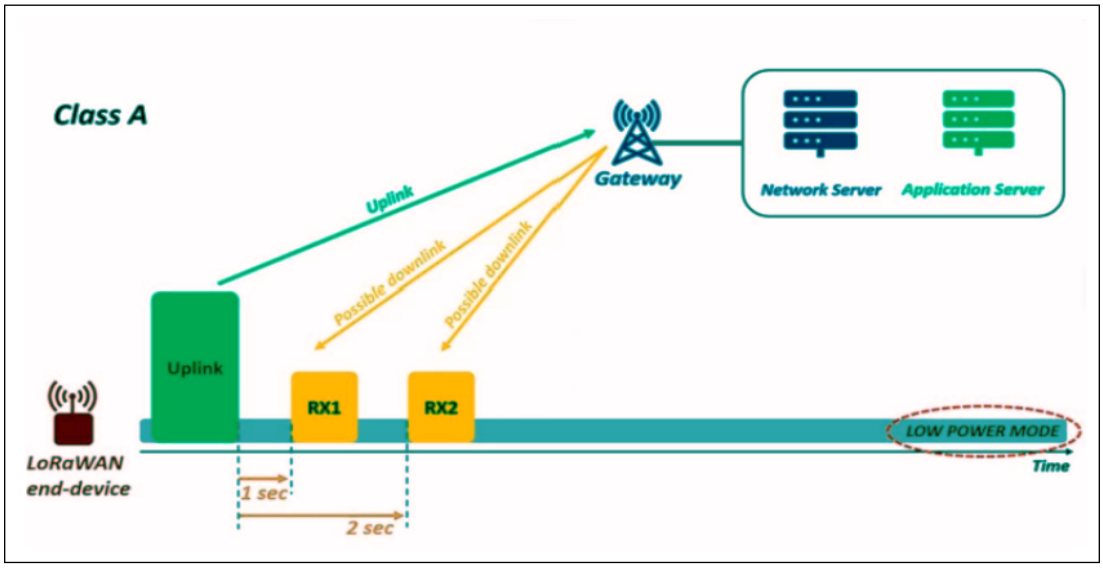


Figure 1.4: Class A End-Device Receive Slot Structure [4]

- **Class B – Scheduled Listening:** Opens extra receive windows synchronized via gateway beacons, balancing power consumption and downlink availability [4] (Figure 1.5).

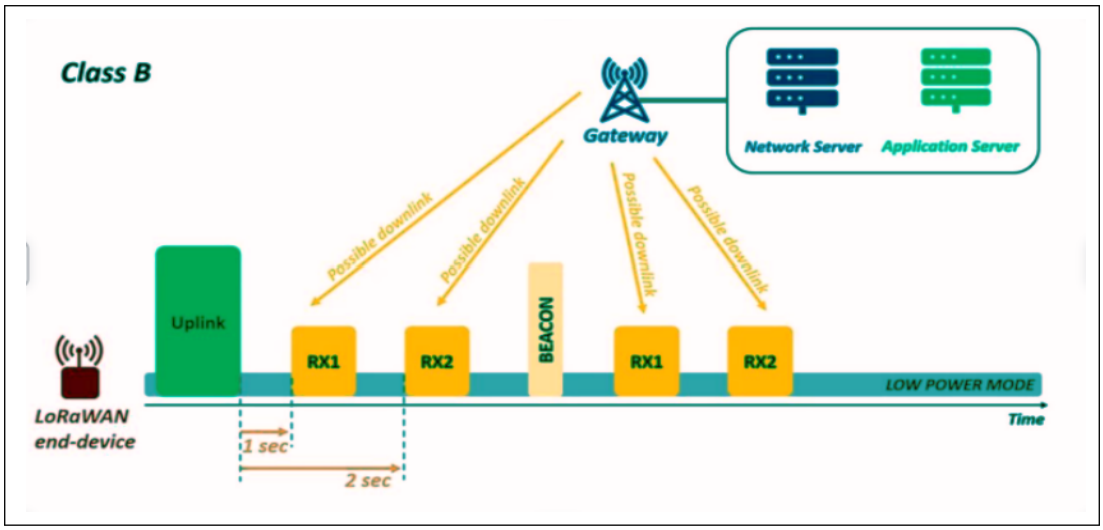


Figure 1.5: Class B End-Device Receive Slot Structure [4]

- **Class C – Continuous Listening:** Devices continuously listen, except during uplinks. This mode provides low latency at the cost of higher power consumption (Figure 1.6).

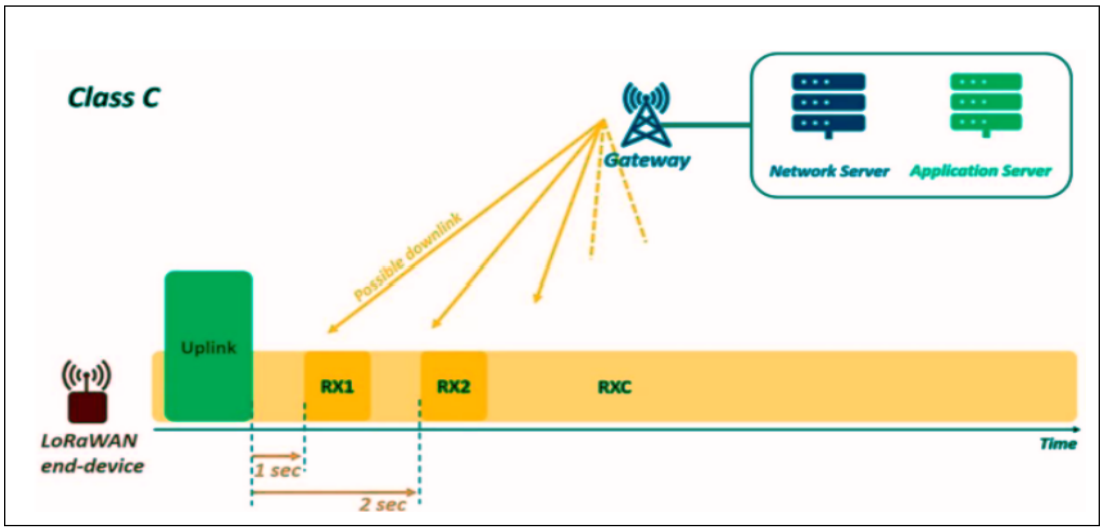


Figure 1.6: Class C End-Device Receive Slot Structure [4]

All devices initially operate in Class A mode by default, while Classes B and C offer enhanced reception capabilities. The choice of class enables LoRaWAN networks to adapt to a wide range of applications, providing a flexible trade-off between energy efficiency and real-time communication [4].

Gateways

LoRaWAN gateways serve as intermediaries between the end devices and the network server. They receive packets over multiple frequencies and SFs, then forward the payloads over IP networks using connections such as Ethernet, Wi-Fi, or cellular as plotted in Figure 1.7.

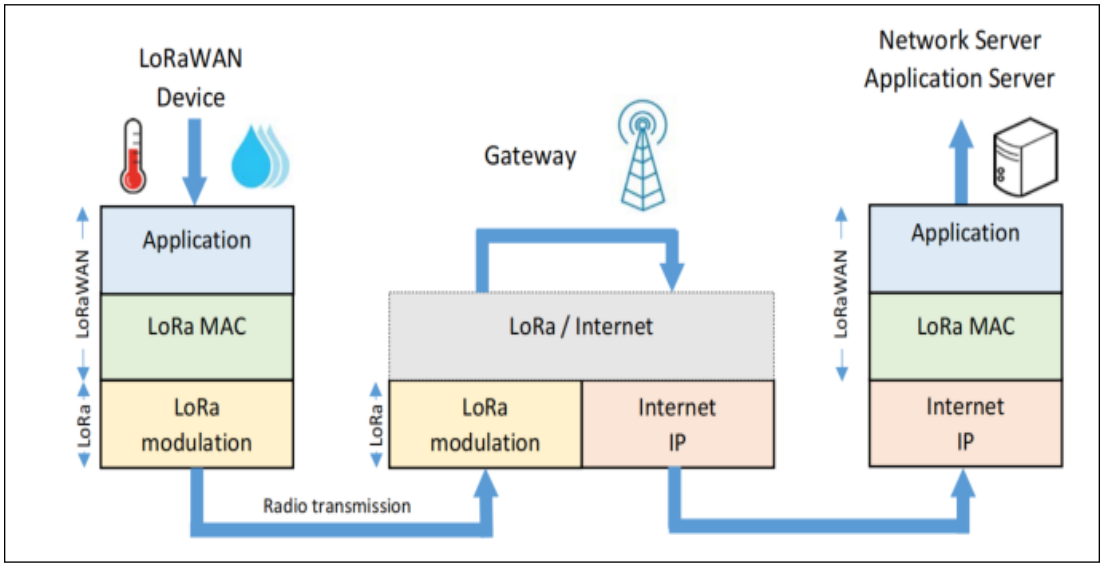


Figure 1.7: LoRaWAN gateway [4]

Each gateway is uniquely identified by a 64-bit EUI, allowing registration and management within the network. Their role is entirely passive: gateways simply relay messages without interpreting them [4].

Network Server

The network server is the intelligence behind the LoRaWAN infrastructure. It aggregates data from multiple gateways, filters duplicates, and authenticates messages using a secure Network Session Key (NwkSKey) as reflected in Figure 1.8.

Only authenticated packets are routed to the application server. This centralized architecture allows for efficient routing, mobility support, and adaptive data rate management.

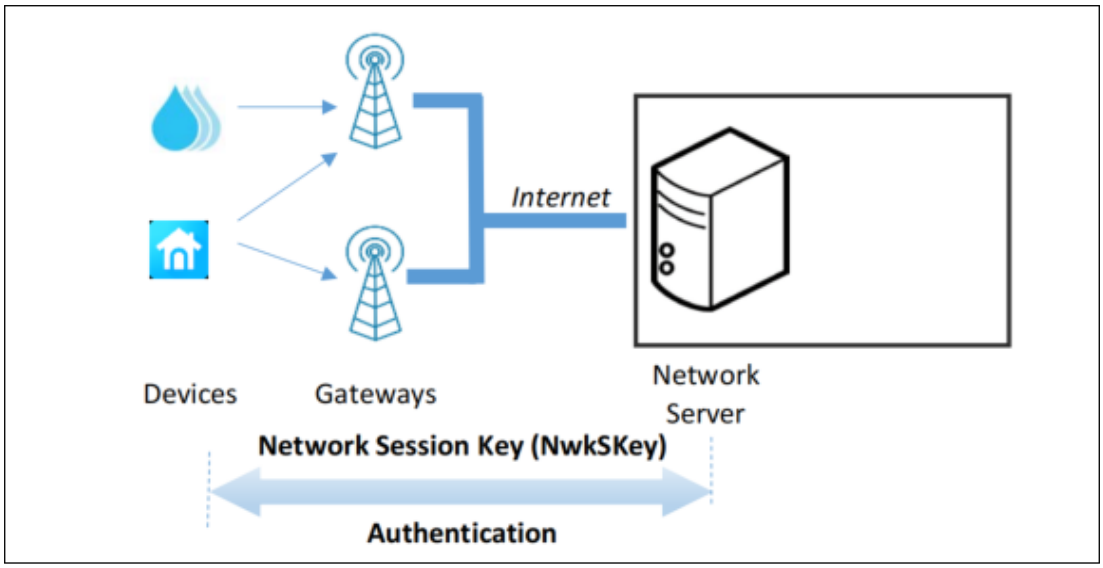


Figure 1.8: Authentication between End-device and Network Server [4]

Application Server

Once messages are authenticated, the Application Server decrypts payloads using the Application Session Key (AppSKey) and delivers usable data to the final user application. It separates sensitive data from network control, preserving confidentiality between the end-device and the application as shown in Figure 1.9.

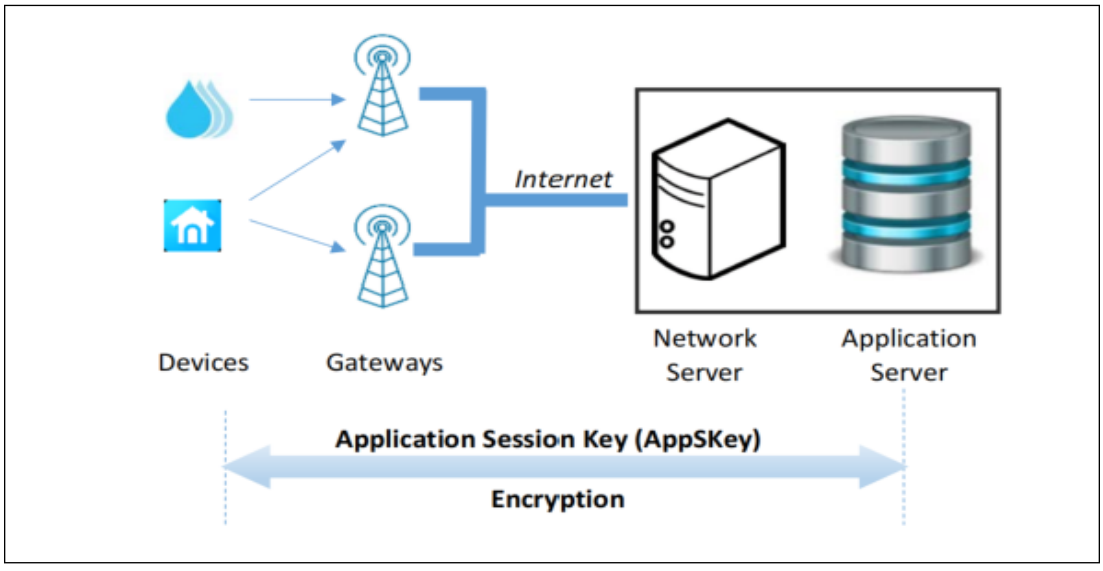


Figure 1.9: Encryption between End-device and Application Server [4]

IoT Platform

The IoT platform represents the user-facing side of the system, integrating data ingestion, storage, and visualization. It typically connects via protocols like HTTP or MQTT and it transforms raw sensor data into actionable insights for end-users (Figure 1.10).

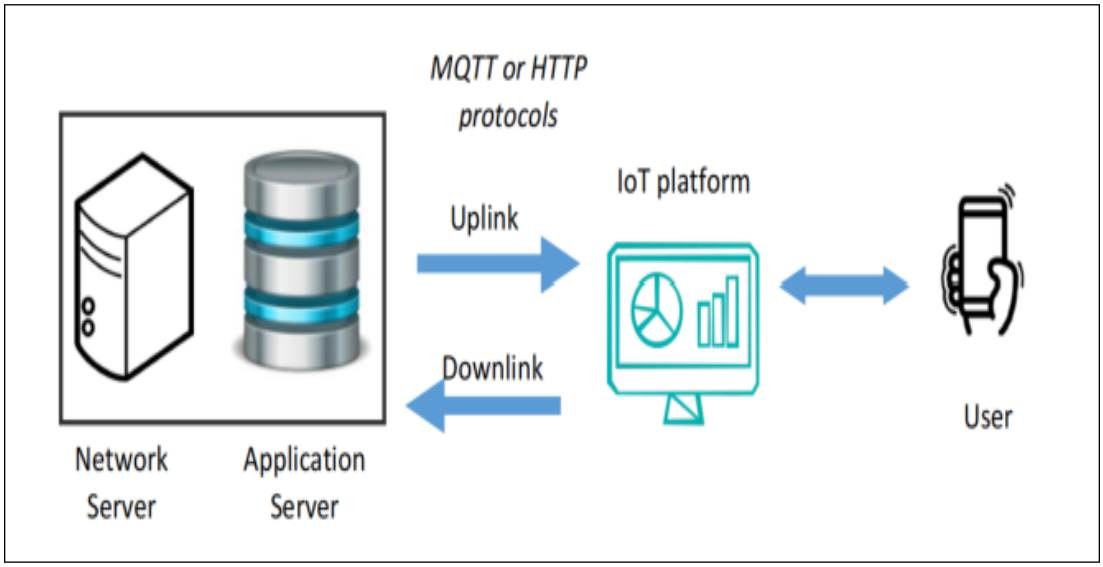


Figure 1.10: Interaction between Application Server and the IoT platform [4]

1.2.2 LoRaWAN: Security Features

LoRaWAN employs a layered security architecture to ensure the confidentiality, authenticity, and integrity of transmitted data, making it suitable for large-scale IoT deployments.

Data Confidentiality, Authentication, and Message Integrity

LoRaWAN ensures data confidentiality through **AES-128**[4] symmetric cryptography. The core of this system relies on two independently managed session keys [4]:

Application Session Key (AppSKey): Encrypts and decrypts the user application data within the **FRMPayload**[5]. This key is shared solely between the end device and the Application Server, preserving strict end-to-end confidentiality.

[4]**AES**: Advanced Encryption Standard a symmetric key algorithm widely used for data protection.
[5]**FRMPayload**: The payload that carries application-specific user data in a LoRaWAN frame.

Network Session Key (NwkSKey): Secures MAC commands and verifies the **Message Integrity Code (MIC)**[6], enabling the Network Server to authenticate messages without accessing application data.

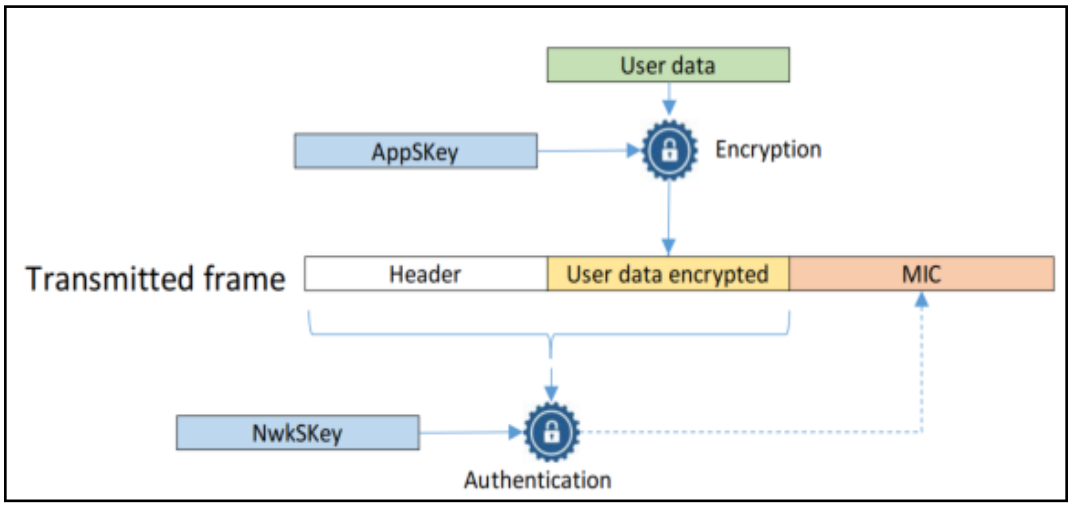


Figure 1.11: Encryption and Authentication process [4]

As indicated in Figure 1.11, this dual-key architecture separates concerns: the AppSKey protects user data, while the NwkSKey secures network-layer operations. Although LoRaWAN uses **AES in ECB mode**[7], it incorporates unique values like the device address and the frame counter to emulate **CTR mode**[8] behavior and prevent repetition. For message integrity and authentication, LoRaWAN applies a cipher-based message authentication code (CMAC) built upon **AES in CBC mode**[9] [3].

LoRaWAN Message Structure and Types

LoRaWAN communication uses a structured message format to ensure secure and flexible data transmission. Each message sent over the LoRa physical layer encapsulates a MAC payload, offering a clear framework for secure exchange. Figure 1.12 shows that every frame includes a physical preamble, a MAC header (MHDR) with an **MType** field that indicates message type, and concludes with a MIC [5].

[6]**MIC**: A cryptographic checksum used to verify message authenticity and prevent tampering.
[7]**ECB mode**: Encrypts blocks independently, which can leak patterns.
[8]**CTR mode**: A block cipher mode using counters to produce non-repeating ciphertexts.
[9]**CBC mode**: Chains ciphertext blocks to improve diffusion and resist certain attacks.

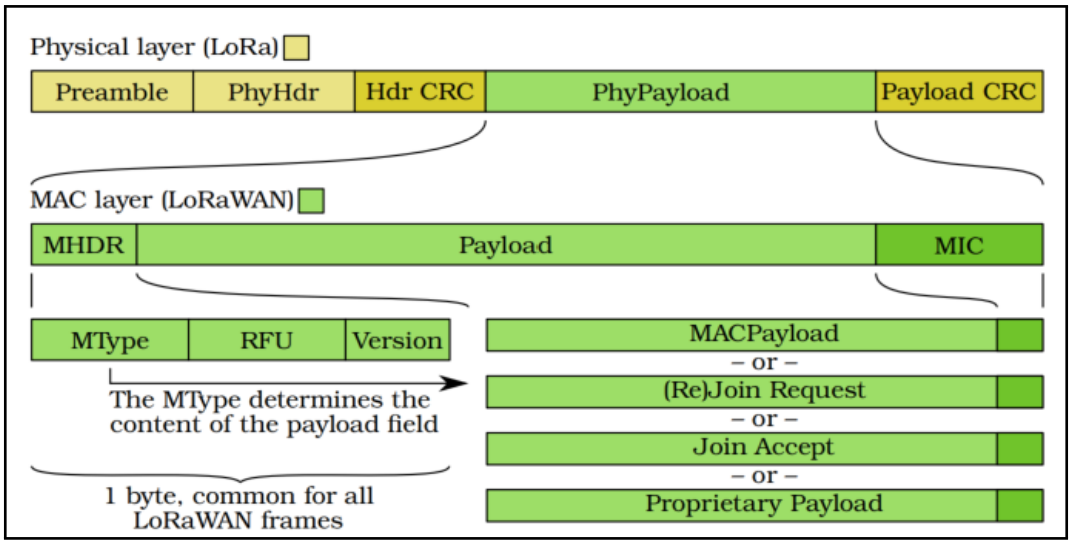


Figure 1.12: Frame structure of LoRaWAN messages [5]

The **MType** field defines the function of each message: join, control, or data transmission. Table 1.1 summarizes the main categories, including Join Request/Accept for onboarding and confirmed/unconfirmed data for ongoing communication [5].

Table 1.1: LoRaWAN Message Types [5]

Bitmask	Type	Transmitter
000	Join Request	End Device
001	Join Accept	Gateway
010	Unconfirmed Data Up	End Device
011	Unconfirmed Data Down	Gateway
100	Confirmed Data Up	End Device
101	Confirmed Data Down	Gateway
110	Rejoin Request	End Device
111	Proprietary	Not Specified

As shown in Figure 1.13, beyond the MHDR, the core of each operational frame lies in the **MACPayload**, that comprises a Frame Header (FHDR), Frame Port (FPort), and application payload (FrmPayload),

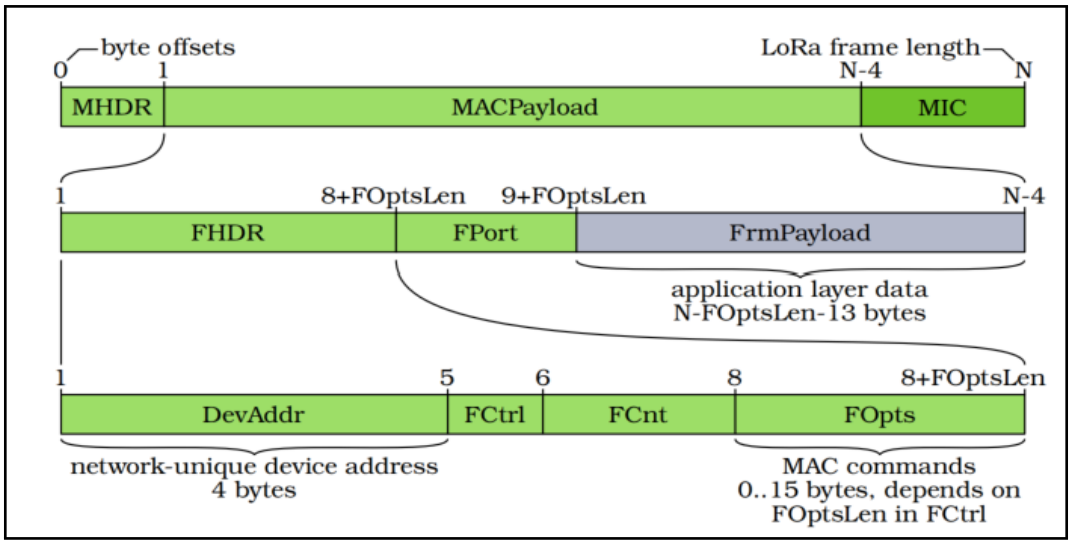


Figure 1.13: Structure of the MACPayload in LoRaWAN messages [5]

The FHDR holds key control data: the device address (**DevAddr** [10]), control flags (**FCtrl**), frame counter (**FCnt**[11]), and optional MAC command fields (**FOpts**[12]). This structure enables features like Adaptive Data Rate (**ADR**[13]), enhancing communication efficiency.

Activation Methods

Before communication begins, devices must authenticate through one of two activation methods: **Activation by Personalization (ABP)** or **Over-the-Air Activation (OTAA)**.

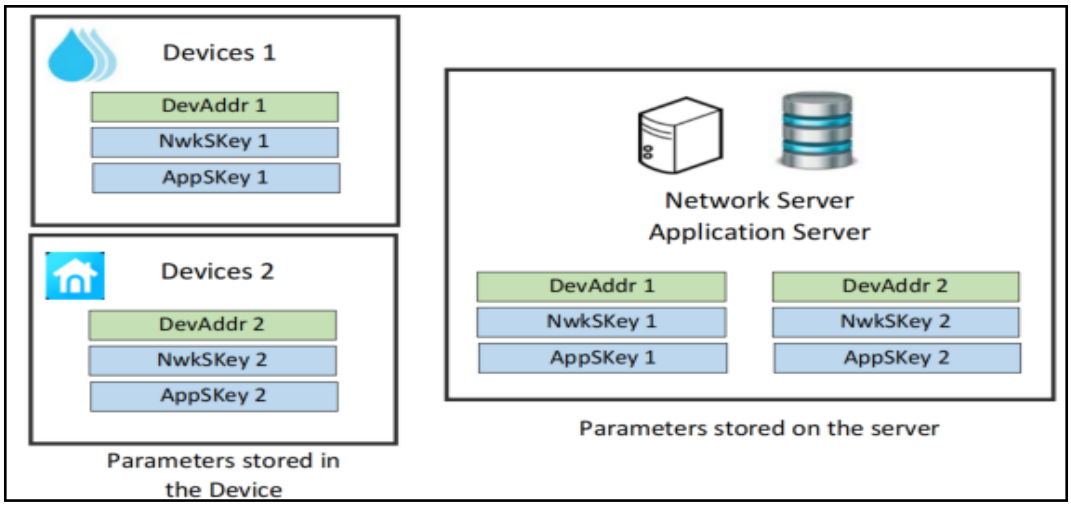


Figure 1.14: Parameters stored in ABP [4]

[10]**DevAddr**: A 32-bit address uniquely identifying the end device within a LoRaWAN network.
[11]**FCnt**: A sequential counter to prevent message replay.
[12]**FOpts**: A field carrying MAC commands for device control and management.
[13]**ADR**: A mechanism for optimizing data rate and power settings based on network conditions.

- **Activation by Personalization (ABP)**: Devices using ABP are pre-configured with static values including **DevAddr**, **AppSKey**, and **NwkSKey**, as shown in Figure 1.14. These devices can transmit immediately upon activation but lack key renewal, making them more vulnerable.
- **Over-the-Air Activation (OTAA)**: OTAA offers better security through dynamic key generation. Devices initiate a **Join Request** with identifiers such as **DevEUI**[14], **AppEUI**[15], and a unique **DevNonce** a random number generated by the device to prevent replay attacks [4].

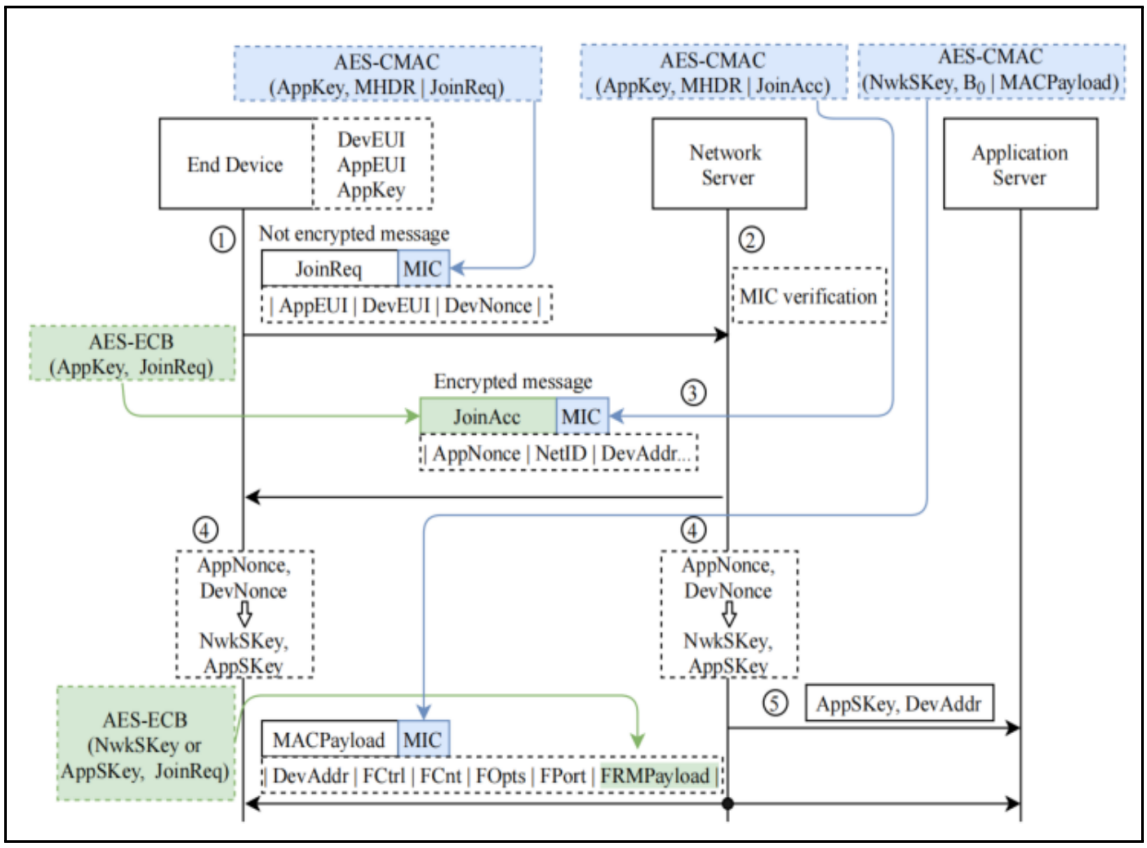


Figure 1.15: OTAA procedure [3]

As detailed in Figure 1.15, the **Join Request** message is not encrypted, but it includes a MIC based on a pre-shared **AppKey**[16] to ensure integrity. The Network Server responds with a signed and encrypted **Join Accept** message containing parameters like **DevAddr**,

[14]**DevEUI**: A globally unique 64-bit identifier assigned to the device by the manufacturer.
[15]**AppEUI**: A globally unique 64-bit identifier that represents application provider associated with the device.
[16]**AppKey**: A root key used to authenticate and derive session keys during OTAA.

AppNonce[17], and **NetID**[18]. Using the **DevNonce** and **AppNonce**, the end device and Network Server dynamically generate the **AppSKey** and the **NwkSKey** [7].

OTAA mitigates also **replay attacks** by tracking used nonces and rejecting duplicates.

1.2.3 LoRaWAN: Security Risks

LoRaWAN's security depends significantly on the chosen activation method. **ABP** is more susceptible to threats since its static keys can be extracted, enabling attackers to decrypt traffic or clone devices. Lack of nonce renewal also increases the risk of replay attacks [7].

OTAA enhances security by generating fresh keys per session, however it introduces trust assumptions: the Network Server creates the AppSKey, potentially exposing application data if compromised. LoRaWAN also remains exposed to broader threats such as **Distributed Denial of Service (DDoS)**[19] and **Man-in-the-Middle (MITM)**[20] attacks, especially if back-end systems are not similarly secured [7].

1.3 Problem Statement

Despite the multi-layered security mechanisms of LoRaWAN, including session-based key separation and OTAA-based dynamic key generation, critical vulnerabilities persist. These weaknesses ranging from static key reuse in ABP to potential exposure of application-layer data underscore the need for rigorous security evaluation.

In this context, we propose the development of **LoRaXploit**, a targeted framework for simulating and analyzing attack scenarios on LoRaWAN-based networks. LoRaXploit aims to uncover and evaluate real-world vulnerabilities through controlled attack simulations, assess their impact on network integrity, and establish a foundation for future security improvements.

[17] **AppNonce**: A server-generated value used to derive session keys in OTAA.
[18] **NetID**: A network identifier that separates LoRaWAN networks.
[19] **DDoS**: An attack that floods a network to disrupt service.
[20] **MITM**: An attack where an adversary intercepts and possibly alters communications between parties.

1.4 Project Objectives

The primary objective of this project is to design and implement **LoRaXploit**, a modular framework for simulating and evaluating security attacks on LoRaWAN-based IoT networks. To achieve this, the project is guided by the following specific goals:

- **Vulnerability Identification:** Analyze the LoRaWAN protocol stack to uncover potential attack surfaces across different layers, including network activation, key management, and data transmission.
- **Attack Simulation:** Develop and execute a suite of controlled attack scenarios such as replay attacks, key re-use exploits, and denial-of-service to assess real-world threat feasibility.
- **Impact Assessment:** Measure the impact of each simulated attack on the confidentiality, integrity, and availability (CIA) of the LoRaWAN system.
- **Security Insight Generation:** Derive actionable insights and technical recommendations for reinforcing LoRaWAN security based on empirical observations from the simulations.

1.5 Work Methodology

For this project, we adopted the **V-Model** as our development methodology. The V-Model offers a structured approach to project management, divided into two main phases as shown in Figure 1.16.

The first phase is the **downward** or development phase, progressing from requirements specification to product implementation. The second is the **upward** or validation phase, which moves from the completed product back through quality assurance and validation activities [8].

Our choice of the V-Model is motivated by the fact that the project requirements are stable and clearly defined, and we must meet a strict deadline with specific milestone commitments. The model's well-defined stages are easy to understand and prepare for in advance.

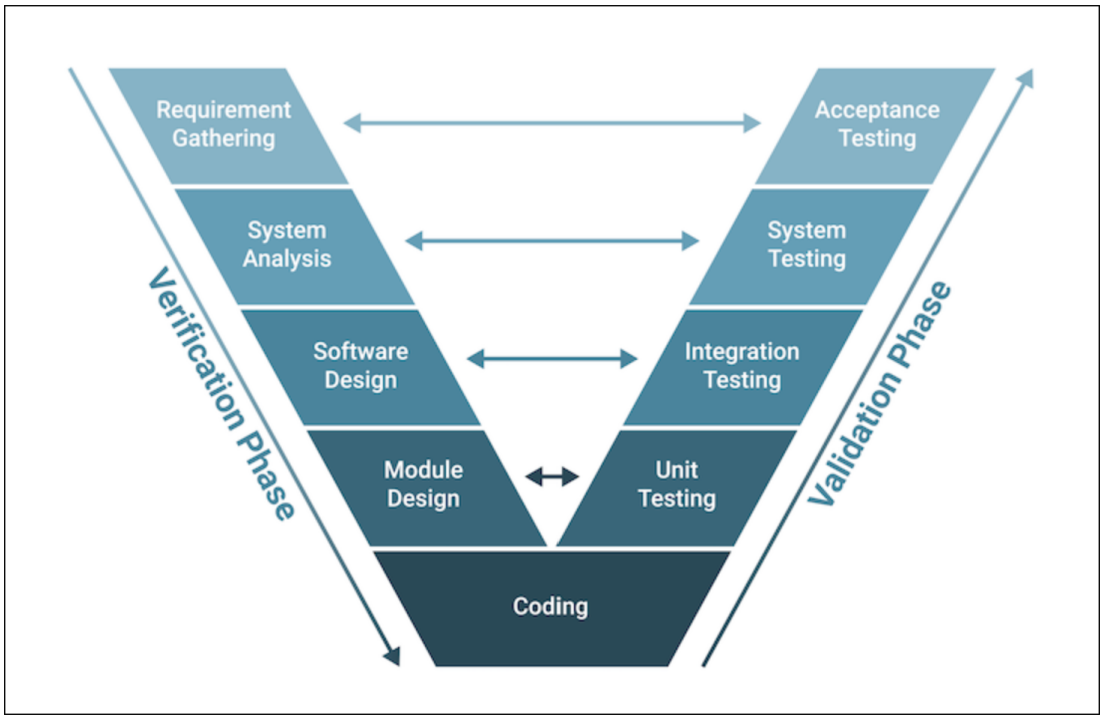


Figure 1.16: V-Model [8]

An important strength of the V-Model lies in its dual-branch structure: each development phase corresponds directly to a related verification or validation step. This interaction between stages ensures continuous quality control throughout the development process and contributes to the delivery of a robust final product.

Conclusion

In this chapter, we explored the fundamentals of LoRaWAN, its architecture, activation methods, and inherent security risks. These insights form the foundation for our proposed attack simulation framework in the following chapters.

Chapitre

2

Vulnerabilities and Security Threats in LoRaWAN

Contents

Introduction

As LoRaWAN becomes a key enabler for IoT connectivity, it increasingly attracts security threats. Despite its secure design, LoRaWAN is vulnerable to various attacks such as replay, eavesdropping, device impersonation, and jamming. This chapter explores the most common vulnerabilities affecting LoRaWAN architectures, setting the stage for the attack simulations and countermeasures developed in the following sections.

2.1 Vulnerabilities in LoRaWAN Architectures

Despite the layered security mechanisms it offers, LoRaWAN especially versions prior to 1.1 has been the subject of extensive research uncovering critical weaknesses. These vulnerabilities impact both the end-device and network server sides, with implications for confidentiality, integrity, availability, and authenticity. In this section, we systematically present the most significant vulnerabilities categorized by protocol mechanisms and architectural layers. We also highlight which specification updates have attempted to address these flaws and where gaps remain [5].

2.1.1 Counter Reset and Overflow

Frame counters in LoRaWAN prevent replay attacks by ensuring message uniqueness. In versions 1.0.x, devices using Activation by Personalization (ABP) were not required to store counters in non-volatile memory (NVM)[21], allowing resets after reboots. This could lead to keystream reuse, compromising confidentiality. Additionally, 16-bit counters were vulnerable to overflow without key changes. LoRaWAN 1.1 introduced 32-bit counters and mandated persistent storage, yet many legacy devices remain exposed [5].

2.1.2 Eavesdropping on MAC Commands

Earlier versions of LoRaWAN transmitted MAC commands in the FOpts field without encryption, revealing sensitive control information such as data rates and frequencies. LoRaWAN 1.1

[21] **NVM**: Retains data across power cycles, critical for maintaining counters.

encrypts these commands, but legacy deployments remain vulnerable to passive surveillance and targeted attacks [5].

2.1.3 Nonce Reuse in OTAA

In LoRaWAN 1.0.x, the randomness and non-repeatability of DevNonce and JoinNonce in the Over-the-Air Activation (OTAA) process were not enforced. This allowed reuse, potentially enabling session key regeneration and data replay. LoRaWAN 1.1 replaced random nonces with monotonically increasing counters to prevent reuse, though inconsistent implementation in older systems continues to pose risks [5].

2.1.4 Join Accept Message Ambiguity

Earlier specifications lacked binding between Join Requests and Join Accept messages, enabling attackers to replay or inject unrelated Join Accepts. LoRaWAN 1.1 resolves this by incorporating nonce values into the Message Integrity Code (MIC), ensuring each response corresponds to its request [5].

2.1.5 Unauthenticated Class B Beacons

Class B devices rely on periodic beacons from gateways to synchronize receive windows. While protected by a CRC[22], these beacons lack authentication, allowing attackers to spoof them. This can desynchronize devices or deplete their batteries. The symmetric key model of LoRaWAN complicates beacon authentication, particularly across untrusted networks [5].

2.1.6 Vulnerable Routing Table Updates

Routing decisions rely on uplink signal quality, making them susceptible to spoofing. An attacker can manipulate signal characteristics to influence gateway selection, redirecting traffic through compromised nodes. Since message origin authentication at the physical layer is absent, this remains exploitable across all LoRaWAN versions [5].

[22] **CRC**: Cyclic Redundancy Check, used to detect errors in transmitted data.

2.1.7 Persistent Vulnerabilities in LoRaWAN 1.1

Despite the notable enhancements introduced in LoRaWAN v1.1 in terms of communication robustness, network availability, and cryptographic improvements the protocol continues to face a range of security and operational challenges. The key vulnerabilities are summarized below [9]:

- **Key Management Deficiencies:** LoRaWAN v1.1 lacks standardized mechanisms for securely rotating root keys or renewing them periodically. Combined with weak key storage practices on constrained devices, this increases the risk of long-term key exposure.
- **Device-Level Physical Vulnerabilities:** End devices often lack tamper protection, secure storage, or trusted execution environments. This enables physical extraction of credentials, especially in unattended or outdoor deployments.
- **Weak End-to-End Trust at Lower Layers:** Although application-layer payloads are encrypted, MAC-layer integrity is enforced only hop-by-hop. This allows semi-trusted network servers to inspect or alter metadata and control messages.
- **Legacy Support Trade-offs:** For backward compatibility with versions 1.0 and 1.0.2, v1.1 inherits design choices that limit the adoption of stricter security postures. This broadens the attack surface and complicates full security modernization.

These limitations highlight that protocol-level improvements alone are insufficient without complementary measures, such as hardware hardening, key rotation strategies, and infrastructure-aware deployment planning [9].

2.2 Attacks and Threats in LoRaWAN

This section outlines the major security threats and vulnerabilities affecting LoRaWAN networks. We categorize the attacks based on their different impacts **authentication**, **availability**, **integrity** and **confidentiality** as observed in Figure 2.1. Then we analyze their effects on network security and reliability.

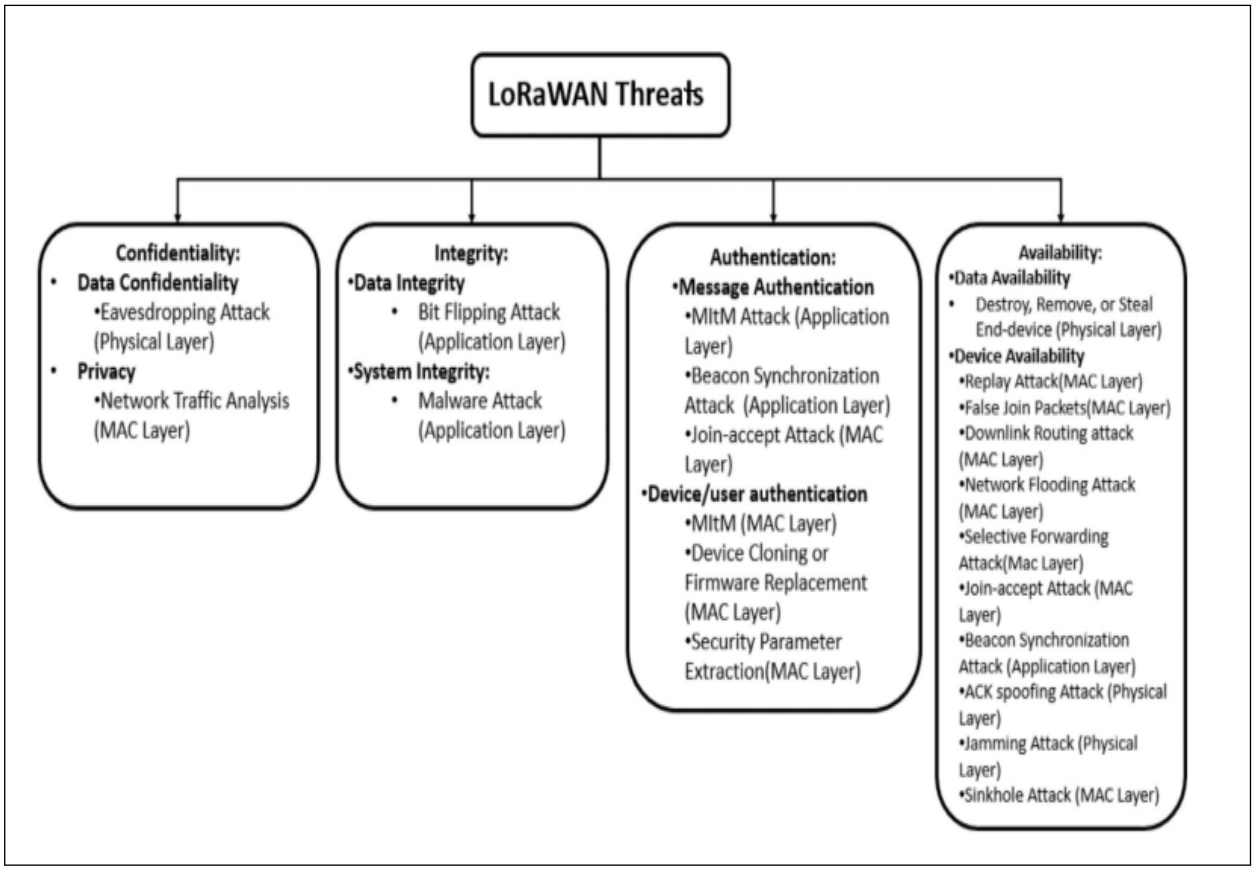


Figure 2.1: LoRawan Threats [9]

2.2.1 Authentication Attacks

LoRaWAN authentication mechanisms are susceptible to several types of attacks, particularly when cryptographic keys are compromised or message validation is insufficient.

Man-in-the-Middle (MITM) Attack

A classic **Man-in-the-Middle (MITM)** attack occurs when an adversary intercepts and alters messages between an **End Device (ED)** and a **Gateway (GW)**. If the attacker possesses the `NwkSKey`, they can modify the payload and recompute a valid **Message Integrity Code (MIC)**, which the **Network Server (NS)** will accept as authentic. This threat becomes more significant during the **Over-the-Air Activation (OTAA)** procedure, where the `AppKey` is used to derive session keys [9].

If an attacker intercepts both the `join-request` and `join-accept` messages, and successfully extracts the `AppKey`, they can derive the `NwkSKey` using the following formula:

```
NwkSKey = AES128(AppKey, 0x01 || AppNonce || NetID || DevNonce || pad16)
```

In this equation:

- `AES128` represents the AES-128 encryption algorithm.
- `AppKey` is the Application Key, which is a shared secret between the **End Device (ED)** and the **Network Server (NS)**.
- `0x01` is a constant value used to indicate that this calculation is for the `NwkSKey` (Network Session Key).
- `AppNonce` is a random value generated during the OTAA process.
- `NetID` is the network identifier.
- `DevNonce` is a unique nonce specific to the device for a particular session.
- `pad16` is a padding value that ensures the input to the AES function is of the correct size.

Once the attacker has derived the `NwkSKey` using this process, they can impersonate the device, inject malicious payloads, and disrupt or eavesdrop on communication between the **End Device (ED)** and the **Network Server (NS)** [9].

Key Extraction and Device Cloning

In scenarios where an attacker obtains physical access to an ED, they may extract cryptographic material from memory or firmware. Devices lacking secure storage or encrypted firmware can be reverse-engineered, allowing an adversary to duplicate the device (i.e., clone it) and impersonate it on the network. This results in duplicated traffic, inflated billing, and potential exposure of private data through legitimate network interfaces [9].

2.2.2 Availability Attacks

Availability is a critical aspect of LoRaWAN, especially for applications relying on timely data delivery. Several attack vectors exist that aim to disrupt or degrade network performance.

Replay Attack

Replay attacks involve capturing and retransmitting valid LoRaWAN messages, particularly **join-request**, **join-accept**, or data uplinks, at a later time. For OTAA, an attacker may exploit the fact that certain fields (e.g., **DevNonce**) may not be validated correctly if reused. In ABP mode, where counters are static or reset upon reboot, replaying old messages becomes feasible. This can cause the NS to misinterpret outdated telemetry, generate unnecessary alarms, or even overwrite more recent data, leading to logical inconsistencies in the application layer [3].

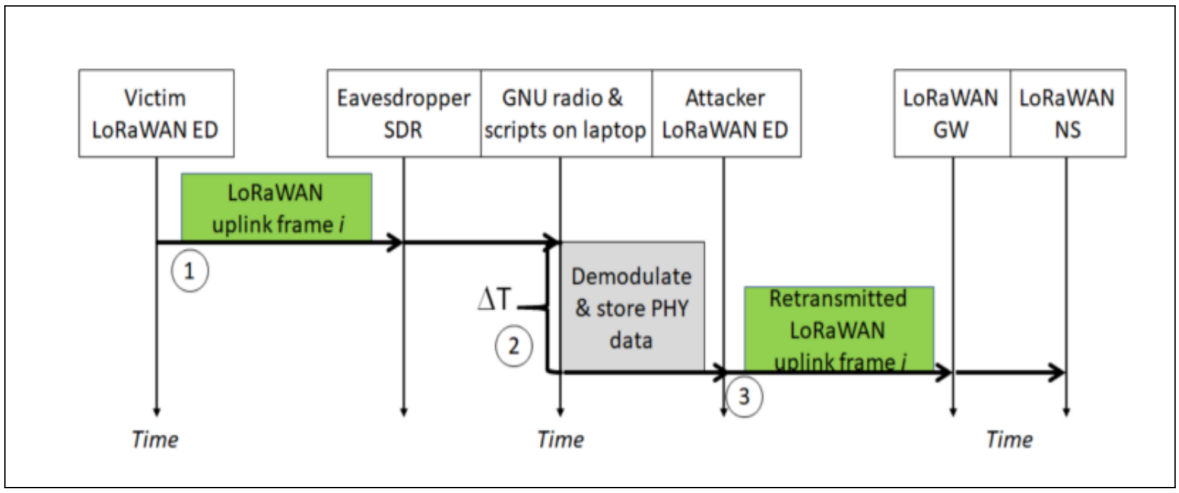


Figure 2.2: Replay attack phases[3]

False Join Packets

Attackers may send forged **join-request** messages using guessed or replayed **JoinEUI**[23] and **DevEUI**. Despite MIC protection, repeated attempts can strain resources or exploit nonce handling, potentially disrupting legitimate sessions [9].

Sinkhole Attack

A malicious node posing as a trusted Gateway can lure traffic by advertising better metrics. Once intercepted, data may be dropped, altered, or exploited. While LoRaWAN does not use routing in the traditional sense, such attacks can still be effective when coupled with rogue backhaul connections to the NS [9].

[23] **JoinEUI:** Identifier for the Join Server during activation.

Jamming Attacks

LoRa's low-power wide-area characteristics make it susceptible to various jamming strategies [9]. As detailed in Figure 2.3 attack types include :

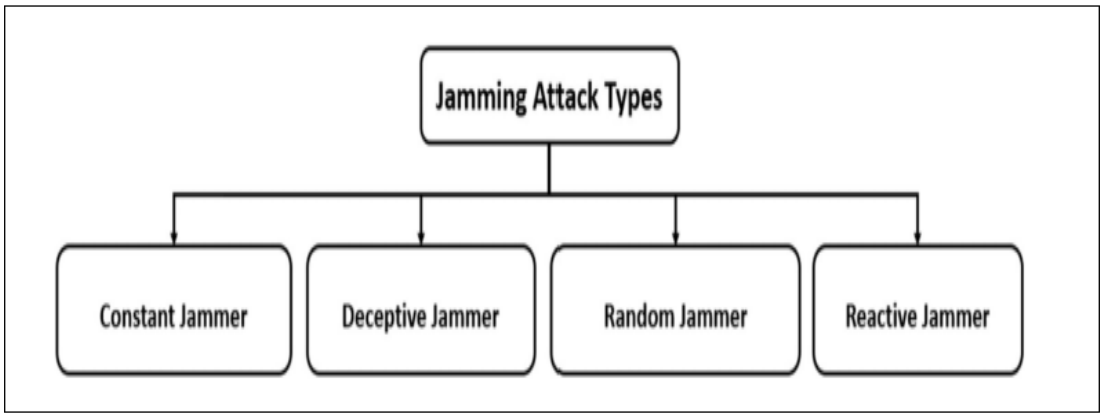


Figure 2.3: Types of Jamming attacks [9]

- **Constant Jammer:** Continuously emits broad-spectrum noise, denying access to all devices in range.
- **Deceptive Jammer:** Sends LoRa-formatted packets with invalid MICs, increasing decoding overhead on Gateways and NSs.
- **Random Jammer:** Alternates between jamming and idle periods to conserve power while still disrupting communication.
- **Reactive Jammer:** Listens for preamble signals and instantly transmits noise during payload transmission, making detection harder and increasing energy efficiency.

2.2.3 Confidentiality Attacks

LoRaWAN ensures confidentiality using AES-128 in counter (CTR) mode, but static key usage and certain implementation flaws expose vulnerabilities.

Eavesdropping and Key-Stream Reuse

LoRaWAN uses CTR-mode encryption, which depends on unique frame counters per session. In ABP mode, these counters can reset after restarts or firmware updates, leading to key-stream

reuse. If two ciphertexts (`C1` and `C2`) share the same key and counter, then :

$$\boxed{C1 \oplus C2 = P1 \oplus P2}$$

If one plaintext is known or guessable (e.g., standard telemetry), the other can be revealed. Even without known plaintexts, structural patterns in telemetry may leak information, compromising confidentiality and possibly enabling command manipulation within the same session [1].

Traffic Analysis and Privacy Attacks

While LoRaWAN encrypts payloads, headers remain in plaintext. Attackers using sniffing tools can capture metadata such as **DevAddr**, frame counters, and timing. These reveal behavioral patterns, device types, or locations. For instance, regular uplinks may indicate smart meters, while sporadic bursts suggest alarms. This leakage allows adversaries to perform profiling, surveillance, or location inference without breaking encryption [9].

2.2.4 Integrity Attack

LoRaWAN ensures message integrity to detect tampering, but weaknesses in its cryptographic implementation can still be exploited. One such method is the bit flipping attack, which abuses the linear nature of XOR-based encryption.

Bit Flipping Attack

LoRaWAN uses AES-128 in CTR mode for encryption. This mode allows controlled modification of ciphertext without decryption, enabling an attacker to craft messages that reveal specific content upon decryption by the Network Server (NS).

As shown in Figure 2.4, an attacker intercepts an uplink message and forges a payload (e.g., `"False"`) by:

- Padding the fake payload.
- Generating the keystream via AES-128 in ECB mode using the known **AppSKey**.

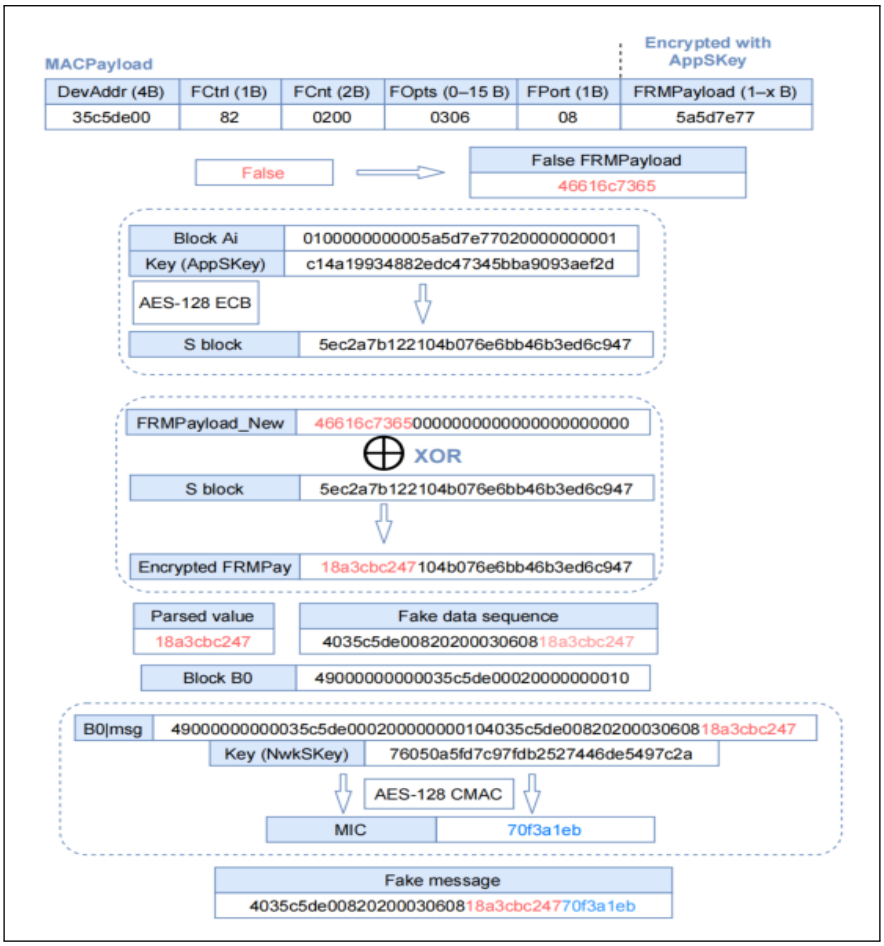


Figure 2.4: Bit Flipping Process [3]

- XORing the keystream with the fake payload to produce new ciphertext.
- Combining the ciphertext with the original header.
- Recomputing the MIC using **NwkSKey** and appending it to the packet.

The NS accepted the forged message because integrity checks only occur at the network layer. The tampered `"False"` payload was treated as valid, revealing the need for end-to-end integrity verification at the Application Server (AS) level.

Conclusion

In this chapter, we analyzed the major security threats and vulnerabilities in LoRaWAN networks. In the next chapter, we will dive into our solution, covering the hardware and software environment, architecture, and its setup.

Chapitre

3

Design and Implementation of a LoRaWAN Security Testbed

Contents

Introduction

This chapter details the setup of our local LoRaWAN testbed used for attack simulations including a GPS-enabled LoRa device, Raspberry Pi gateway, ChirpStack servers, and visualization tools like InfluxDB and Grafana. We also outline the network architecture and provide installation and configuration steps for a fully operational security evaluation system.

3.1 Work Environment

Our experimentation is conducted within a locally hosted LoRaWAN environment designed for flexibility and control. The setup integrates both hardware and software components essential for running and evaluating attack scenarios.

3.1.1 Hardware Environment

In our prototype, we leverage both a high performance host PC for running backend services and a Raspberry.Pi-based gateway kit paired with End device development boards. This approach enables us to delegate radio packet forwarding and gateway bridging to the Pi, while managing ChirpStack servers, InfluxDB, and Grafana on the PC, ensuring a scalable and modular testbed.

Table 3.1: Hardware Components

Component	Specifications	Role
Host PC	Windows-11 Pro, Intel-Core-i7 (8-core), 32-GB RAM, SSD storage	Runs containerized ChirpStack services : ChirpStack Network & Application Servers, MQTT broker, InfluxDB, Grafana.
Raspberry-Pi-3	Broadcom BCM2837, 1-GB RAM, Quad-core 1.2-GHz CPU	Hosts the Semtech packet forwarder and acts as the physical host for the gateway module (RHF0M301)[10].

Table 3.1 – Hardware Components (continued)

Component	Specifications	Role
Gateway Module (RHF0M301–868)	868-MHz ISM band, Semtech SX1301 concentrator (8 demod paths), SMA	RF front-end for LoRaWAN; sends/receives LoRa packets [10].
PRI-2 Bridge (RHF4T002)	SPI-to-GPIO level translator	Interfaces Pi's SPI pins to the RHF0M301 module reliably [10].
Seeeduino LoRaWAN w/ GPS (RHF76-052AM)	ATmega328P + SX1276 transceiver, u-blox-compatible GPS, micro-USB	Functions as a LoRaWAN End-device; enables uplink of sensor data and GPS location tracking. Arduino-compatible and programmable via Arduino IDE [10].

3.1.2 Software environment

Our prototype leverages a mix of LoRaWAN-specific services, general IoT tooling, development languages, and system utilities to build, monitor, and pentest the network.

Table 3.2: Software Components

Component	Logo	Role & Configuration
Docker		Containerized deployment for ChirpStack, Mosquitto, PostgreSQL, InfluxDB & Grafana [11].
Raspberry.Pi.OS		Host OS on the Pi, runs packet forwarder & ChirpStack Gateway Bridge [12].
WSL (Ubuntu-24.04.LTS)		Linux CLI on Windows-11 PC for installing/testing Linux-only tools [13].

Table 3.2 – Software Components (continued)

Component	Logo	Role & Configuration
ChirpStack (Gateway Bridge, Network Server & Application Server)		Single container hosting all ChirpStack components; handles packet bridging (UDP→MQTT), LoRaWAN MAC management, ADR, frame decryption, device/app management, and REST API [14].
PostgreSQL		Backend database for ChirpStack Network Server (stores device sessions, frame counters, ADR state) [15].
Mosquitto MQTT Broker		Message bus connecting Gateway Bridge AND ChirpStack services [16].
InfluxDB		Time-series database for sensor data, with retention policies configured [17].
Grafana		Dashboards for real-time and historical visualization of metrics [18].
LoRaWAN Auditing Framework (LAF)		Toolkit to craft, parse, send, analyze, and crack LoRaWAN packets for security auditing and pentesting [19].
Python		Scripting for data processing, automation tasks, attack simulation scripts [20].
C		Firmware development on Seeeduino end-nodes and low-level packet tools [21].
Arduino.IDE		Environment for writing/uploading Seeeduino sketches and managing libraries [22].

Table 3.2 – Software Components (continued)

Component	Logo	Role & Configuration
Wireshark		Packet capture and protocol analysis (Ethernet, MQTT, SPI via extenders) [23].
PuTTY		SSH/Telnet client for remote management of Raspberry.Pi and other devices [24].

3.2 Targeted Architecture

This section outlines the overall architecture of our experimental LoRaWAN security testbed.

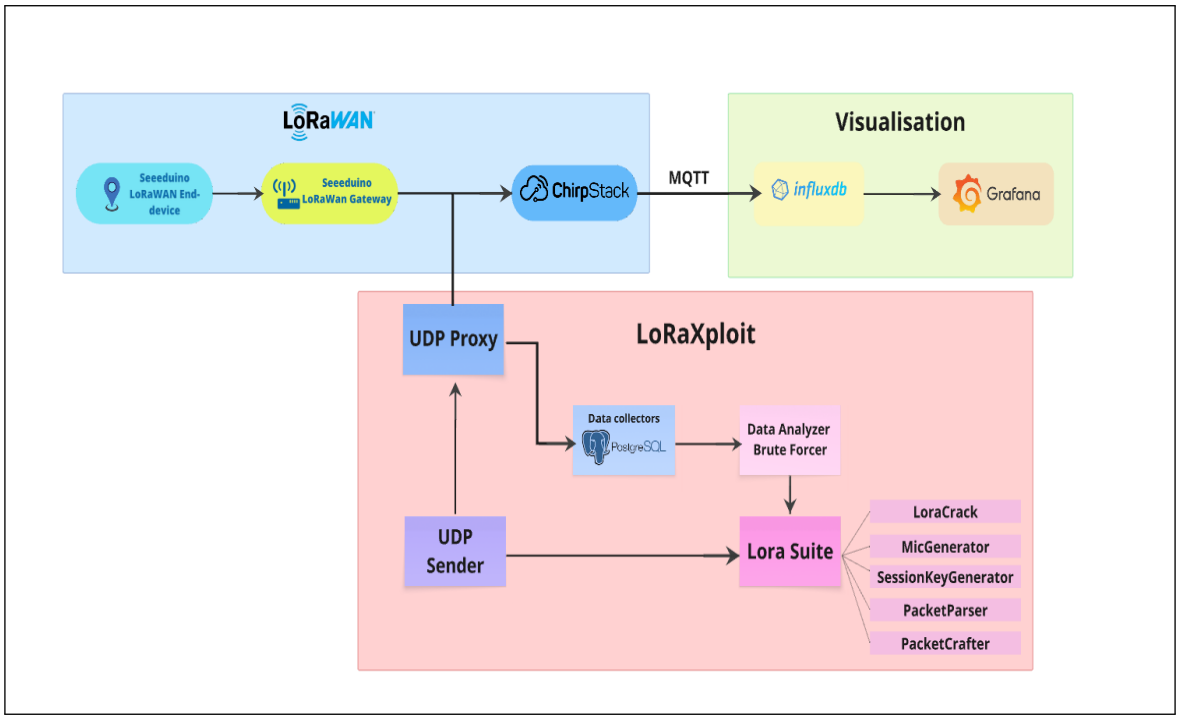


Figure 3.1: Targeted Architecture

The design connects a Seeeduino-based end device, a Raspberry Pi-powered LoRaWAN gateway, and services including ChirpStack, PostgreSQL, InfluxDB, and Grafana. In parallel, it integrates our custom framework, **LoRaXploit**, which enables traffic interception, data analysis, and targeted attack simulation through a modular suite of tools as demonstrated in Figure 3.1. Below, we detail each functional block and its role in the testbed workflow.

3.2.1 End Device & Gateway

The system begins with a Seeeduino LoRaWAN GPS-enabled end device, responsible for generating real telemetry data. This data is wirelessly transmitted to the Seeeduino LoRaWAN Gateway, which forwards packets using the Semtech UDP Packet Forwarder protocol. The gateway serves as the bridge between the physical LoRa interface and the software-defined network.

3.2.2 ChirpStack Network & Application Server

ChirpStack acts as both the Network Server (NS) and the Application Server (AS) in our setup. It receives LoRaWAN packets from the gateway and handles decryption, MIC validation, frame counter tracking, and payload decoding. As the AS, it also publishes decoded application data to external consumers via MQTT, facilitating integration with the visualization and monitoring stack.

3.2.3 Visualization Stack (InfluxDB & Grafana)

To visualize application-level data, MQTT output from ChirpStack is ingested by InfluxDB, a time-series database optimized for storing telemetry. Grafana then queries InfluxDB to render real-time dashboards that display sensor readings, signal quality, and other network metrics. This makes the testbed suitable for both performance monitoring and anomaly detection.

3.2.4 LoRaXploit Framework

LoRaXploit operates as a parallel control plane for targeted security testing. It intercepts and processes raw LoRaWAN packets via a UDP Proxy, stores relevant metadata in a PostgreSQL database, and feeds the packets into a modular analysis pipeline. The framework supports traffic manipulation, session key derivation, MIC forgery, and injection of crafted packets into the network.

UDP Proxy & UDP Sender

The UDP Proxy monitors the bidirectional traffic between the gateway and ChirpStack, enabling deep packet inspection and selective manipulation. The UDP Sender is responsible for injecting forged or modified packets back into the network, allowing the execution of active attacks such as replay, injection, and bit-flipping.

Data Collectors & Analyzer

Captured packet metadata is stored in a PostgreSQL database for querying, replay, and offline analysis. The analysis module includes a brute-force engine designed to recover the **AppKey** used during the OTAA join procedure by testing candidate keys against observed MICs. It also detects key reuse patterns and reconstructs session contexts, enabling the crafting of targeted attack scenarios.

LoRa Suite (Toolkit Modules)

At the core of LoRaXploit lies the **LoRa Suite**, a modular collection of tools:

- **LoRaCrack**: Brute-forces the AppNonce in JoinAccept messages to derive session keys by validating MICs using the known **AppKey**.
- **MicGenerator**: Computes valid MICs for forged application payloads.
- **SessionKeyGenerator**: Derives session keys (**AppSKey/NwkSKey**) from OTAA join parameters.
- **PacketParser**: Parses LoRaWAN frames and extracts MAC and application layer fields.
- **PacketCrafter**: Builds well-formed LoRaWAN packets for injection, based on inferred session state.

3.3 Environment Setup

This section details the setup of a modular LoRaWAN test environment, covering the installation of the gateway, end device, ChirpStack stack, and LoRaXploit framework for security testing

and attack simulations. The installation process for the gateway, end device, and ChirpStack stack follows the instructions provided in the repository [25].

3.3.1 Hardware Setup

In our project, we began by connecting the Gateway module **RHF0M301-868** to the **PRI 2 Bridge (RHF4T002)**, which was then plugged into the **Raspberry Pi 3** as shown in Figure 3.2.

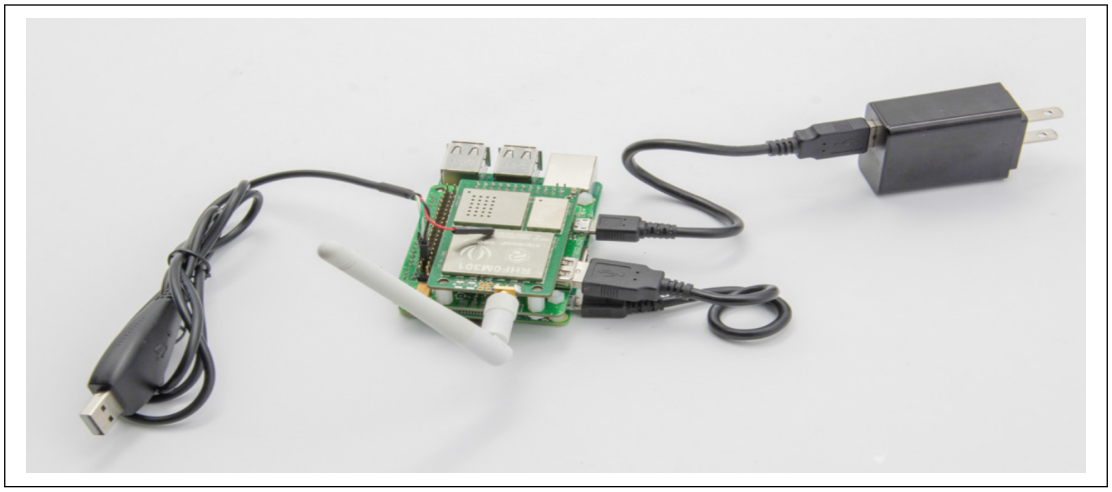


Figure 3.2: Connection of the Gateway module to the PRI 2 Bridge and Raspberry Pi 3

We configured the Raspberry Pi with Wi-Fi and enabled both **UART** and **SSH** connections as as indicated in Figure 3.3, allowing us to access the system via **PuTTY** (see Figure 3.4).

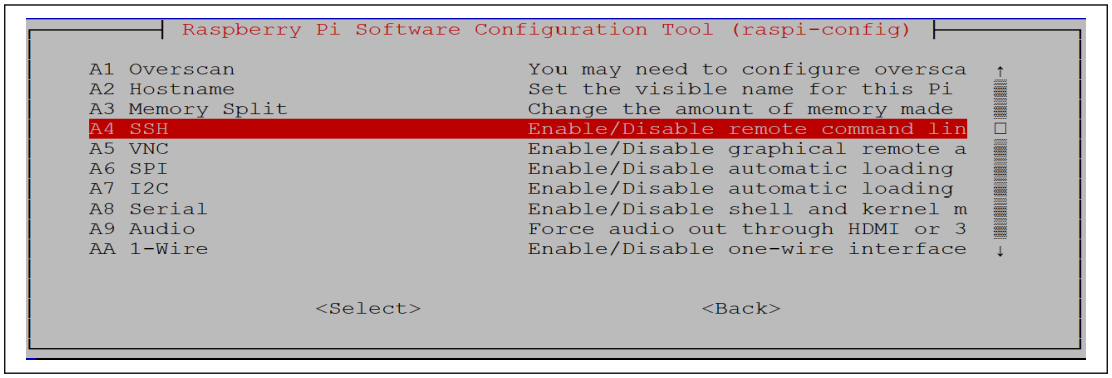


Figure 3.3: Raspberry Pi Configuration

Once connected, we expanded the SD card file system and enabled the **pktfwd** service to forward packets from the gateway to the server.

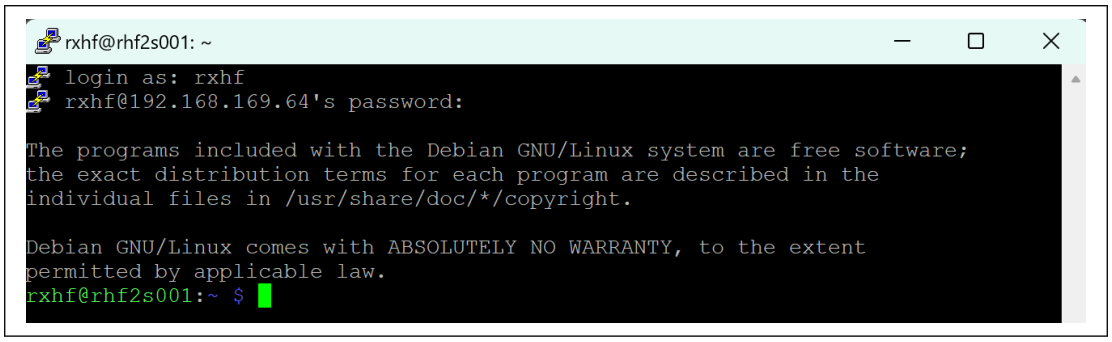


Figure 3.4: PuTTY Configuration for Raspberry Pi access

Next, we installed the **Arduino IDE** on our PC to interface with the **Seeeduino LoRaWAN with GPS module (RHF76-052AM)** (shown in Figure 3.5). Using the IDE, we uploaded a custom Arduino sketch [26] to the Seeeduino module.

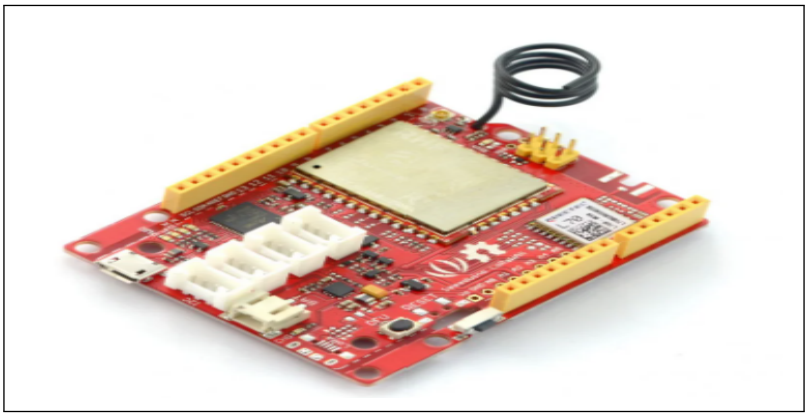


Figure 3.5: Seeeduino LoRaWAN modem with GPS (RHF76-052AM) [10]

This sketch enabled serial communication for sending AT commands, allowing us to retrieve the device identifiers **DevEUI** and **DevAddr**, as demonstrated in Figure 3.6.

Figure 3.6: Result of AT commands showing DevEUI and DevAddr

These identifiers were later used in the **ChirpStack server** configuration to securely register the device and establish communication within the LoRaWAN network.

3.3.2 ChirpStack Installation and Configuration

To enable secure and scalable LoRaWAN communication, we deployed **ChirpStack** on the host PC using Docker Compose, as shown in Figure 3.7. The stack integrates essential components such as the Network Server, Application Server, and a PostgreSQL/MQTT backend. This configuration allows seamless communication between our gateway and the application layer.

Figure 3.7: Chirpstack docker UP

Gateway Configuration:

The gateway was configured to forward LoRaWAN packets using the Semtech UDP protocol. We modified its configuration to point to the ChirpStack server's IP address and ensured proper uplink/downlink port settings. The gateway's unique identifier (gateway_ID) was preserved for registration in the ChirpStack interface as observed in Figure 3.8. After configuration, the system was rebooted to apply the changes.

Figure 3.8: Gateway Configuration for packets forwarding to ChirpStack Server

ChirpStack Web UI Setup:

Once the gateway was network-ready, we accessed ChirpStack's web interface and performed the following:

- Created a new Network Server to handle routing.
- Added a Service Profile to define the transmission characteristics.
- Registered the gateway using its ID, verifying its active status on the dashboard.
- Created an Application to group and manage end devices.
- Added the Seeeduino LoRaWAN device to the application using its unique identifiers.

During the device registration process, an `AppKey` was generated for OTAA (Over-The-Air Activation), which was required for the firmware setup on the end device.

End Device Integration:

The Seeeduino LoRaWAN module was configured using the Arduino IDE to support OTAA and GPS data transmission using this sketch [27]. Once deployed, the node successfully joined the ChirpStack network, with live data flowing into the application dashboard, confirming end-to-end communication as observed in Figure 3.9.

Figure 3.9: ChirpStack Dashboard: the registered gateway and active device session

3.3.3 Database and Visualization Stack

To effectively store, monitor, and analyze data transmitted by LoRaWAN end devices, we implemented a robust backend composed of **InfluxDB**, **Grafana** and **Mosquitto MQTT**.

a) MQTT Integration:

We configured ChirpStack to publish decrypted application data to an **MQTT broker**. This setup served as the bridge between the ChirpStack Application Server and InfluxDB, allowing real-time data to be captured and stored efficiently.

b) Time-Series Storage with InfluxDB:

InfluxDB was chosen to store the GPS telemetry and radio parameters such as **RSSI**[24] and **SNR**[25]. Its time-series data capabilities allowed us to efficiently manage and query sensor updates over time as represented in Figure 3.10.

```
Using database chirpstack
> SELECT * FROM "chirpstack_data"
name: chirpstack_data
time                applicationID applicationName devEUI           deviceName deviceProfileName             fCnt fPort
host        object_latitude object_longitude topic                                        txInfo_dr txInfo_frequency
----                ------------ ------------- -----           ---------- ---------------             ---- -----
----        -------------- --------------- -----                                        ---------- ---------------
1744153613744313213 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 12   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         868500000
1744153628605050350 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 13   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         867100000
1744153643745830807 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 14   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         868100000
1744153658576794818 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 15   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         868300000
1744153673736274624 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 16   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         867700000
1744153688561207444 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 17   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         867900000
1744153703718453832 2            GPS             8cf9572000031d70 GPS_device OTAA_cayenne_PP_payload_device 18   8
23bdfef2503d 0              0               application/2/device/8cf9572000031d70/event/up 0         867500000
```

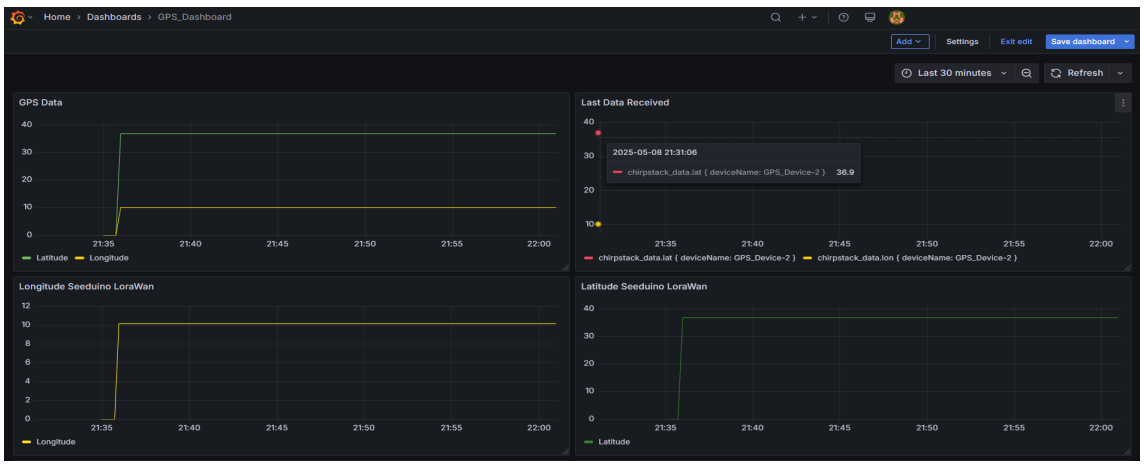
Figure 3.10: Integration of ChirpStack with Mosquitto MQTT and InfluxDB

[24]**RSSI**: A measure of the power level received by a radio frequency (RF) device.
[25]**SNR**: Signal-to-Noise Ratio, a measure of the signal strength relative to the background noise.

c) Visualization with Grafana:

We integrated **Grafana** with InfluxDB to build a dynamic dashboard that visualizes essential metrics. As evidenced by Figure 4.12, the dashboard displays:

- Real-time **longitude and latitude** data captured by the GPS module.
- **RSSI** (Received Signal Strength Indicator) to assess signal power.
- **SNR** (Signal-to-Noise Ratio) to evaluate link quality.
- **Fcnt** (Frame Counter) to monitor the number of transmitted packets.



(a) Real-time visualization of GPS location

(b) Visualization of RSSI, SNR, and Fcnt

Figure 3.11: Grafana Dashboard: Real-time visualization

This visualization enables us to monitor node behavior, detect coverage issues, and understand device mobility through geospatial representation.

3.3.4 LoRaXploit Framework Setup

To simulate and evaluate potential vulnerabilities in our LoRaWAN deployment, we prepared a containerized environment for the **LoRaXploit** framework. This setup ensured modularity, isolation, and ease of dependency management across different attack tools.

All necessary dependencies were installed within the container, enabling a fully operational environment for running and customizing attack simulation scripts. The framework was configured with key modules including the **UDP Proxy**, **UDP Sender**, **Analyzer**, and **LoRa Suite**, each designed to target specific components of the LoRaWAN communication stack.

This environment lays the foundation for the penetration testing phase of our project. In the following chapter, we will detail each attack scenario executed using LoRaXploit, including methodology, tools, and results, providing a comprehensive assessment of the system's resilience.

Conclusion

This chapter outlined the hardware and software setup of our LoRaWAN environment, detailing the system architecture and the preparation of the LoRaXploit testing framework. With the environment fully configured, we are now ready to explore attack simulation scenarios and evaluate effective security measures in the next chapter.

Chapitre

4

Attack Simulation Scenarios and Security Measures

Contents

Introduction

This chapter presents a practical attack simulation against a LoRaWAN network. The scenario follows a structured methodology, beginning with passive traffic interception, proceeding through cryptographic key extraction, and culminating in the injection of a forged packet to desynchronize a legitimate device. All the scripts used in this attack have been uploaded to the Git repository of our project [28]. We will conclude by proposing effective countermeasures to mitigate such threats in real-world deployments.

4.1 Step 1: Traffic Interception (Man-in-the-Middle Attack)

The simulation begins with passive interception using **UdpProxy** [29], a tool configured to operate as a transparent proxy between the LoRaWAN gateway and the network server. This setup enables real-time monitoring and logging of all UDP traffic exchanged between the two endpoints, including both uplink and downlink LoRaWAN messages critical for subsequent stages of the attack. This approach constitutes a classic **Man-in-the-Middle (MitM) attack**, in which the adversary silently observes communications without disrupting the data flow.

a) Objective

The goal is to collect raw LoRaWAN packets in real time from legitimate devices, particularly **JoinRequest** and uplink packets. These contain essential values such as the **DevEUI**, **DevNonce**, and **MIC**, which are required for deriving session keys and decoding payloads in later phases of the attack.

b) Process Overview

In standard LoRaWAN deployments, UDP port **1700** is used by gateways to communicate with the ChirpStack network server. In our test environment, we introduced UDP port **1702** to intercept specific uplink messages without interfering with normal traffic.

As shown in Figure 4.1 **UdpProxy** was configured to receive packets from the gateway on port 1702 and transparently forward them to ChirpStack on port 1700, maintaining uninterrupted communication.

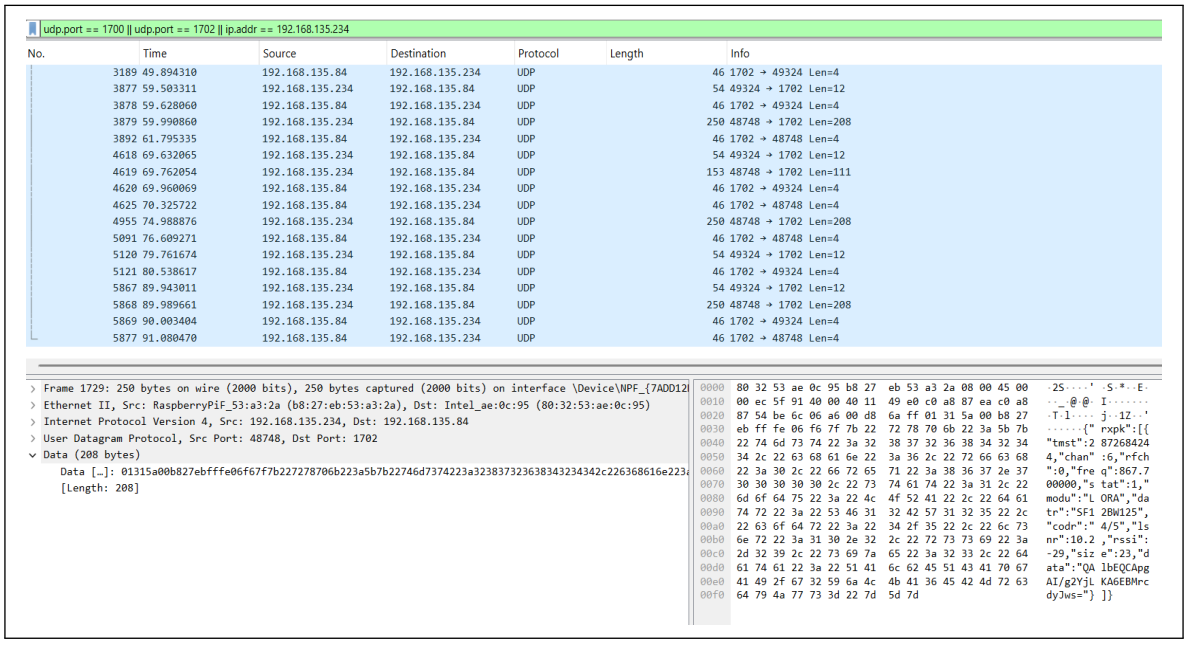


Figure 4.1: Wireshark capture showing UDP traffic on ports 1700 and 1702

Simultaneously, it duplicated the intercepted packets to a local data collection service on port **1800** (see Figure 4.2).

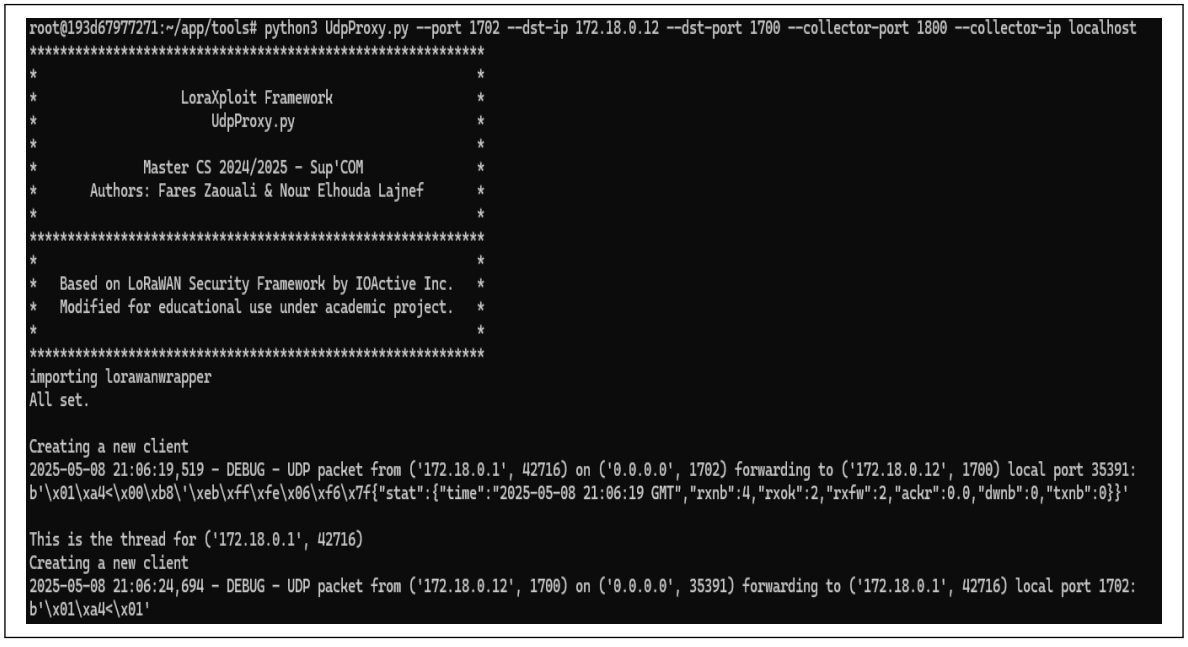


Figure 4.2: Interception and forwarding of LoRaWAN packets

This architecture provided both real-time visibility into LoRaWAN traffic and persistent storage for offline analysis, enabling subsequent cryptographic operations such as brute-force attempts and session key derivation.

c) Outcome

The intercepted traffic yielded a rich dataset of **Join Procedure** as illustrated in Figure 4.3 and session-level messages. These included critical identifiers such as **DevEUI**, **DevNonce**, and **MIC**. This dataset served as the basis for subsequent cryptographic attacks, highlighting the practicality and threat of Man-in-the-Middle (MitM) surveillance within LoRaWAN networks.

Figure 4.3: Join Request and Join Accept packets

4.2 Step 2: Data Collection and AppKey Brute-Forcing

Captured packets were processed using **XploitProcessData**, a tool that integrates key modules of the LoRaXploit framework. At this stage, we focused on analyzing stored **JoinRequest** messages to understand packet structure and launch brute-force attacks against weak **AppKey** implementations.

a) Objective

To recover the 16-byte **AppKey** used by the target end-device. This key is crucial for decrypting JoinAccept messages and deriving the session keys (**NwkSKey** and **AppSKey**) required for authenticated communication.

b) Process Overview

JoinRequest messages, previously intercepted by **UdpProxy** and forwarded to a collection interface on port 1800, were first handled by the **PacketForwarderCollector.py** [30] script as represented in Figure 4.4.

Figure 4.4: Terminal output of PacketForwarderCollector

This collector:

- Listened for UDP traffic in Semtech Packet Forwarder format.
- Parsed the LoRaWAN **PHYPayload**[26] using **PhyParser** module.
- Extracted metadata including RSSI, SNR, and timestamps.
- Persistently stored parsed packets for later analysis.

Stored packets were then processed by **XploitProcessData.py** [31], which includes three modules:

- **Analyzer**: Categorized packets and extracted metadata patterns.
- **Bruteforcer**: Performed AppKey brute-force by testing dictionary keys against JoinRequest MICs.
- **Parser**: Decoded valid payloads into readable formats.

[26]**PHYPayload** is the top-level structure in LoRaWAN packets, containing the MAC header (MHDR), MACPayload, and Message Integrity Code (MIC).

The brute-force process relies on the cryptographic integrity of the JoinRequest **MIC**, which is derived from the AppKey. If a candidate key generates a MIC that matches the one in the intercepted packet, it can be confidently identified as the correct AppKey due to the collision resistance of the MIC algorithm.

c) Outcome

As shown in Figure 4.5, this phase successfully recovered the AppKey of the target device. This marked a pivotal breakthrough in the attack, enabling full control over uplinks and downlinks of the impersonated node.

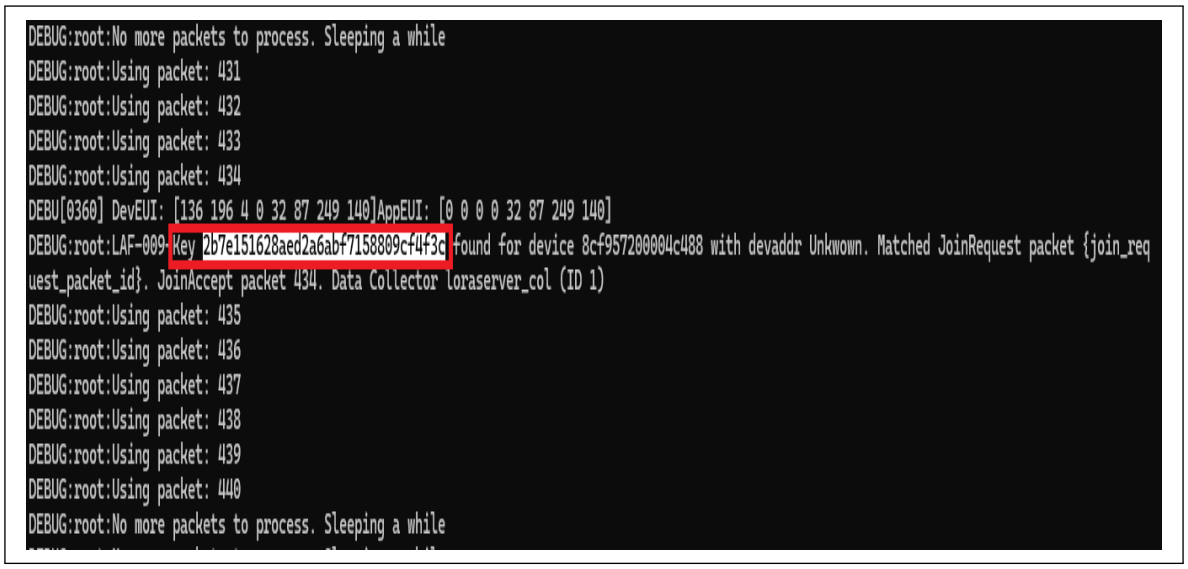


Figure 4.5: AppKey successfully brute-forced

4.3 Step 3: Extracting DevNonce and Deriving Session Keys

With the AppKey successfully recovered from intercepted JoinRequest packets, the next phase in the attack simulation involved computing the session keys required to decrypt traffic and impersonate the end device. This was achieved using the **LoraCrack** [32] toolkit.

a) Objective

To derive the two core session keys:

- **NwkSKey** (Network Session Key): used to compute and verify the Message Integrity Code (MIC).

- **AppSKey** (Application Session Key): used to encrypt and decrypt the application payloads.

These keys are essential for forging valid LoRaWAN messages and decrypting legitimate ones.

b) Process Overview

The derivation process began with identifying all necessary elements to compute session keys: **AppKey** recovered through brute-force, **DevNonce**, **JoinEUI** and **DevEUI** that were extracted from the Join Request.

However, in addition to these fields, the derivation of session keys requires the **AppNonce**, a 3-byte random value generated by the network server during the JoinAccept message. Since this value is not present in the JoinRequest, we performed an exhaustive search over the **AppNonce** space using **LoraCrack**.

The cracking process, as shown in Figure 4.6, involved several coordinated steps to efficiently recover valid session keys:

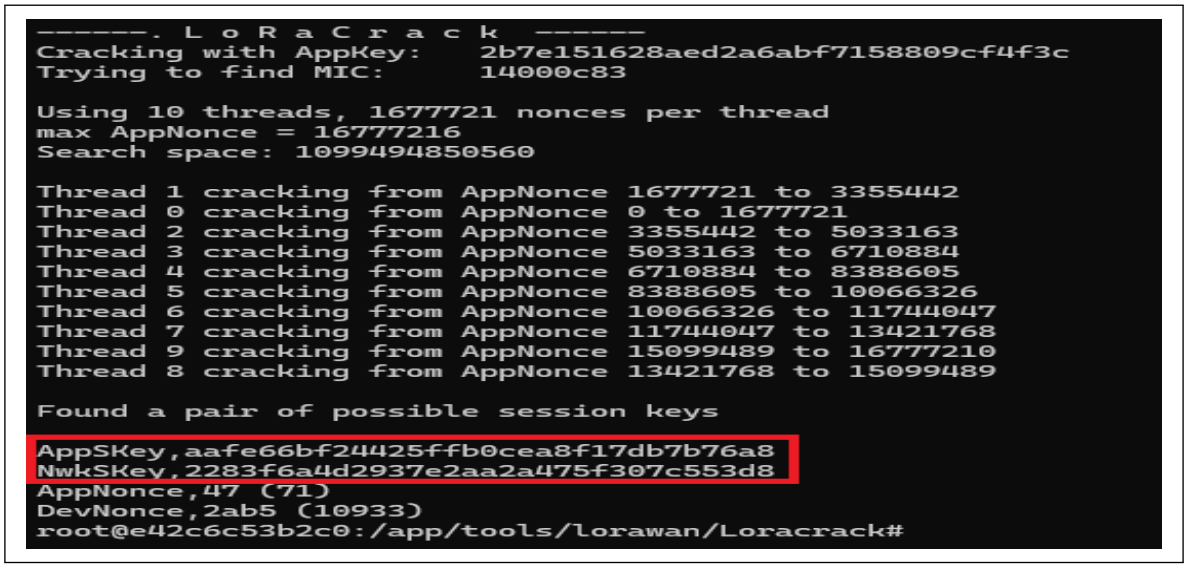


Figure 4.6: AppSKey and NwkSKey successfully cracked

- Iteratively testing possible **AppNonce** values over the entire 3-byte range.
- Deriving candidate session keys (**AppSKey** and **NwkSKey**) for each guessed AppNonce using the recovered AppKey and extracted DevNonce.
- Validating each candidate key set by computing the Message Integrity Code (MIC) of a known data packet and comparing it to the intercepted MIC.

The search was parallelized to accelerate the cracking process through **multi-threading**. Once a valid MIC was reproduced, the correct AppNonce was confirmed, and the corresponding session keys were successfully derived.

c) Outcome

This step marked a turning point in the simulation, transitioning from passive eavesdropping to active manipulation. With the session keys in hand, we gained the same cryptographic privileges as the end-device itself. This formed the foundation for subsequent decoding, packet forging, and injection attacks.

4.4 Step 4: Payload Decryption

Having successfully derived the session keys and intercepted encrypted uplink messages, the next phase focused on decoding these packets to reveal their application-level data. For this task, we utilized **PacketParser.py** [33] to analyze the base64-encoded **PHYPayloads** and extract their human-readable content.

a) Objective

To decrypt the **FRMPayload**[27] of legitimate packets using the previously cracked **AppSKey**, in order to access and understand the transmitted application data such as GPS coordinates.

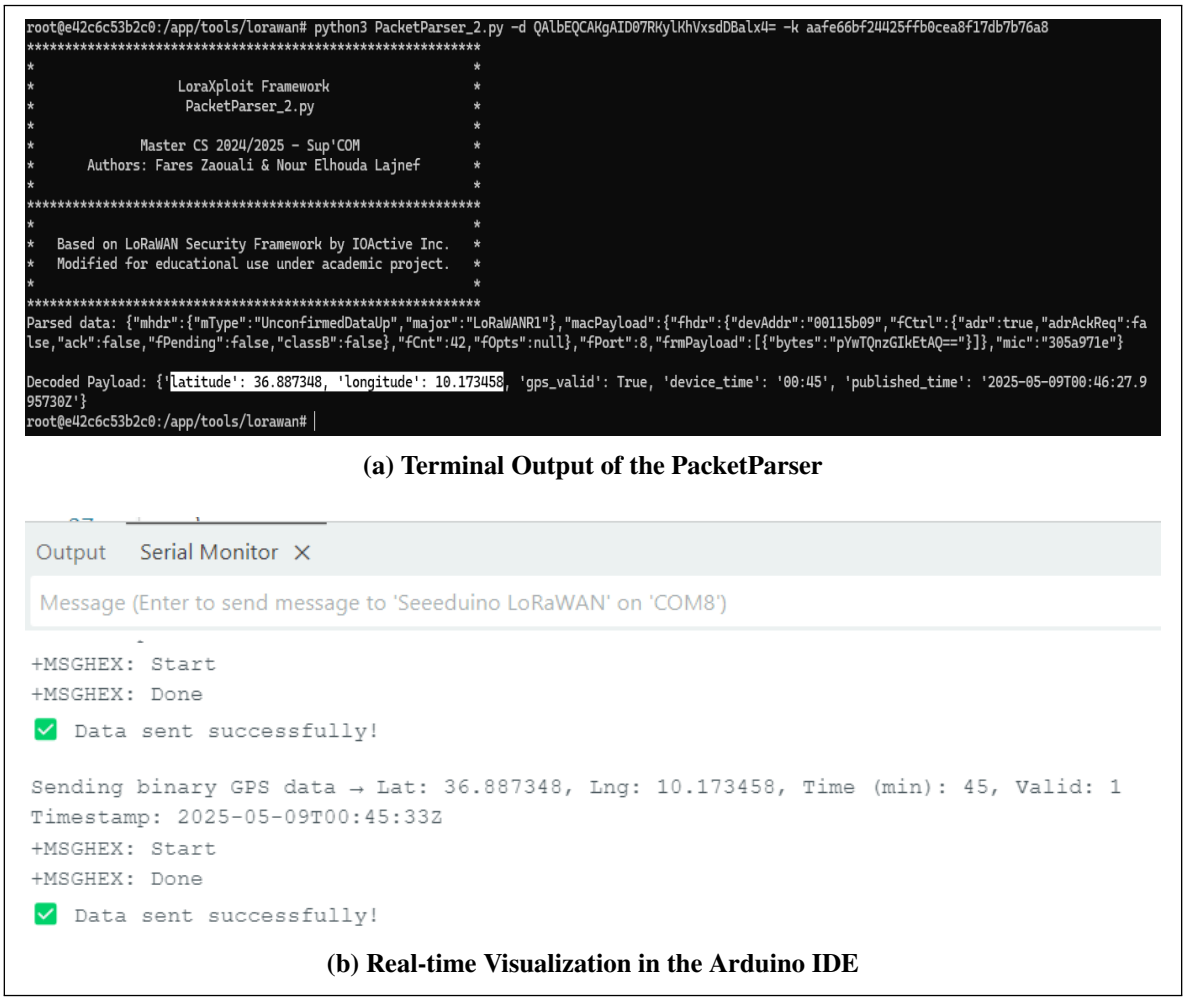
b) Process Overview

PacketParser operates on base64-encoded LoRaWAN packets captured from the air using earlier tools. The parser:

- Decodes the full LoRaWAN structure, including MHDR, FHDR, FCnt, and MIC fields.
- Uses the AppSKey to decrypt the FRMPayload, which carries the encrypted application layer message.
- Reconstructs the plaintext message and metadata for further inspection or replay.

[27] **FRMPayload**: The portion of a LoRaWAN packet that carries application-layer data, encrypted using the AppSKey.

To further enhance the output, a dedicated **Decoder module** was appended to the parsing pipeline. This module translated the decrypted byte stream into semantically meaningful content such as latitude, longitude, device time, and validity flags. The result was a precise and intelligible snapshot of the data that the legitimate device was sending to the network as shown in the Figure 4.7.



(a) Terminal Output of the PacketParser

(b) Real-time Visualization in the Arduino IDE

Figure 4.7: Visualization of parsed LoRaWAN packets

c) Outcome

This step exposed the full structure and content of uplink payloads, including sensitive values like GPS coordinates and device timestamps. It validated the correctness of the derived session keys and confirmed that we could decrypt and interpret messages exactly as the legitimate server would.

4.5 Step 5: Crafting a Forged LoRaWAN Packet

Building on the extracted session keys and decoded payload structures, we proceeded to generate a malicious but structurally valid LoRaWAN message. This step combined GPS data encoding and LoRaWAN frame construction to prepare a spoofed packet intended for injection in the next phase.

a) Objective

To construct a syntactically and cryptographically valid LoRaWAN packet that mimics a real device but manipulates key fields, particularly the frame counter (**FCnt**), in preparation for a denial-of-service attempt.

b) Process Overview

The packet crafting process simulated the behavior of a legitimate LoRaWAN end-device in the following sequence:

- GPS coordinates were encoded into the application-specific 10-byte binary format using **Encoder.py** [34], mimicking real sensor readings as represented in Figure 4.8

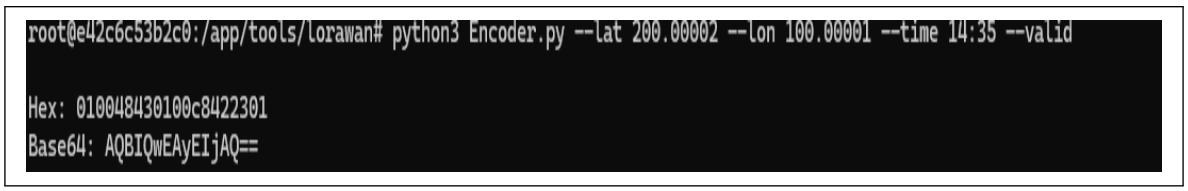


Figure 4.8: Encoding Fake GPS coordinates

- This binary sequence formed the **FRMPayload**.
- Using **PacketCrafter.py** [35], we then assembled the full LoRaWAN frame (**PHYPayload**) by injecting:
  - The **AppSKey** and **NwkSKey** to encrypt the payload and compute the MIC.
  - The known **DevAddr** to align with the target device.
  - A deliberately inflated **FCnt** value (e.g., 500) far ahead of the last observed legitimate counter (e.g., 122) as indicated in Figure 4.9.

Figure 4.9: Crafting malicious Packet

c) Outcome

This operation successfully produced a valid, base64-encoded LoRaWAN packet with a spoofed payload and elevated frame counter.

4.6 Step 6: Injecting the Forged Packet with UdpSender

a) Objective

To simulate a legitimate LoRaWAN uplink transmission using a forged packet in order to execute a desynchronization-based denial-of-service (DoS) attack. This step aimed to validate the crafted payload and session keys by injecting a high **FCnt** packet that would override the network server's internal state and disrupt further communication from the actual device.

b) Process Overview

We employed **UdpSender.py** [36] to transmit a base64-encoded, malicious **PHYPayload** over UDP in the Semtech Packet Forwarder format. The forged packet was directed to `1702`, the default port for gateway-to-server communication via the UDP proxy as visualized in Figure 4.10.

Figure 4.10: Injection of forged uplink message

Key technical aspects included:

- Use of the legitimate **DevAddr** and correctly derived session keys (**AppSKey** and **NwkSKey**) to ensure a valid **MIC**.
- Setting the **FCnt** (frame counter) to `500`, significantly higher than the last known legitimate value (e.g., `130`), to intentionally desynchronize the server's state as plotted in Figure 4.11.

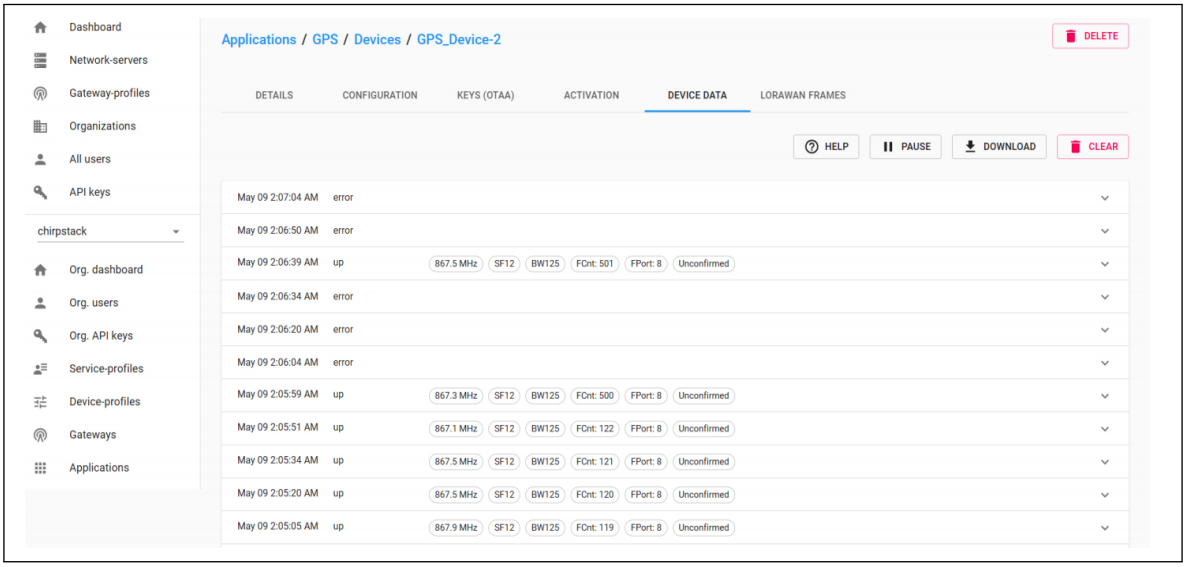


Figure 4.11: Desynchronization of Chirpstack due to the attack

- Inclusion of realistic LoRa physical layer parameters such as frequency **867.3 MHz** and spreading factor **SF12BW125**, ensuring the payload mimicked a real uplink.
- Full compliance with the LoRa gateway bridge format used by ChirpStack, allowing seamless injection via the UDP proxy.

c) Outcome

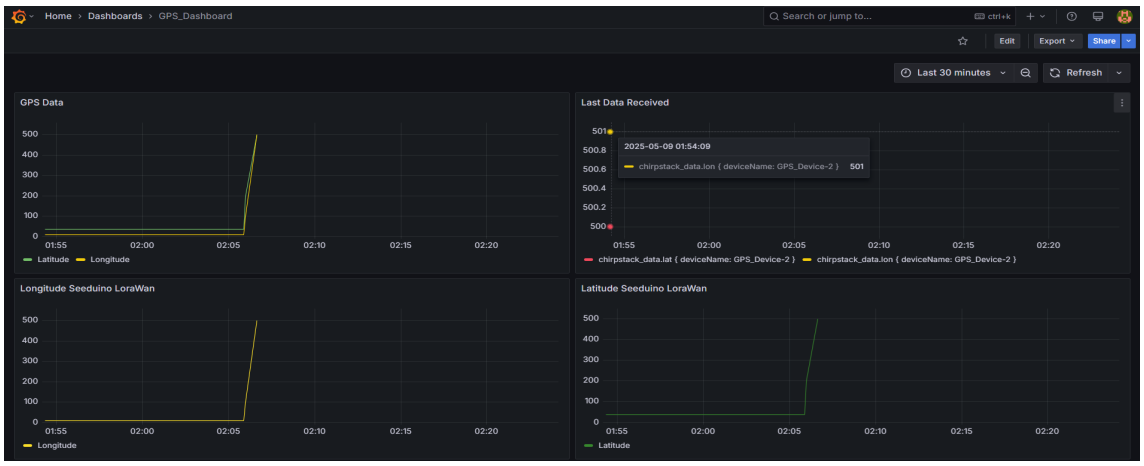
The ChirpStack server accepted the forged packet without issue, as it passed all integrity checks and matched expected metadata. Critically, the server updated its internal frame counter for the device to **500**. As a result, any subsequent legitimate uplink messages with a lower **FCnt** were silently rejected.

To confirm the attack's success, we immediately sent another forged packet with **FCnt = 501**, as reflected in Figure 4.11, which was also accepted by the server. At this point, only attacker-controlled packets could be processed, while the legitimate device unaware of the manipulation continued sending uplinks that were systematically discarded by the server.

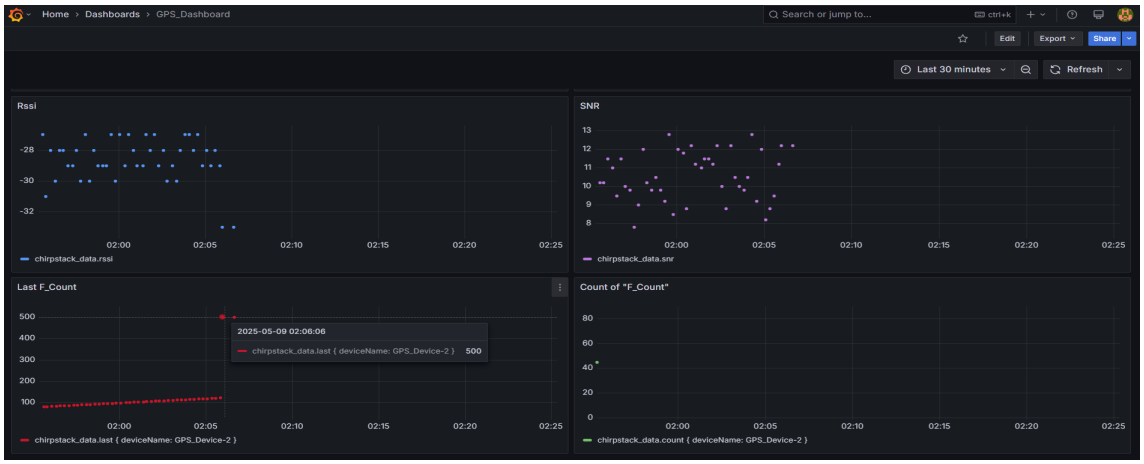
The effects of this desynchronization attack were clearly observable through the Grafana monitoring dashboard. Figure 4.12 presents post-attack data visualization:

- In Subfigure 4.12a, the GPS values were altered in the attacker-crafted payloads—first to `Lat: 200, Lng: 300`, and then to `Lat: 500, Lng: 500`. These forged values replaced the actual location data (`Lat: 36.887348, Lng: 10.17345`), successfully spoofing the monitored telemetry.
- In Subfigure 4.12b, the frame counter progression clearly illustrates the attack timeline. The legitimate counter reached `FCnt = 122`, followed by an abrupt jump to `500`, then `501` corresponding to our two injected packets. From that point onward, no new data was received from the original device, confirming the effectiveness of the desynchronization.

This desynchronization attack showcases how LoRaWAN's dependence on monotonically increasing frame counters for replay protection can be subverted.



(a) Real-time visualization of GPS location after attack



(b) Visualization of RSSI, SNR, and FCnt showing desynchronization

Figure 4.12: Effects of injected malicious packets on location and frame counter values

The result is a stealthy, persistent DoS attack that requires no RF jamming or physical access. Moreover, this method enables other threats such as:

- Replay of previously captured packets.
- Injection of malformed or spoofed sensor data.
- Protocol fuzzing for network-side robustness testing.

Ultimately, this step not only validated the forged packet's correctness but demonstrated a practical exploit against a live LoRaWAN deployment.

4.7 Step 7: Security Recommendations and Measures

To outline practical, effective countermeasures that network operators, developers, and system integrators can adopt to mitigate the threats demonstrated specifically replay attacks, packet injection, and desynchronization via **FCnt** manipulation.

a) Key Protection and Management

- **Use Secure Elements (SE) or Hardware Security Modules (HSMs):** Store root keys (AppKey, NwkKey) in tamper-resistant hardware to prevent key extraction and session key prediction.
- **Avoid Default or Vendor-Provided Keys:** Replace default or publicly known keys with strong, cryptographically secure random values during device provisioning.
- **Key Rotation and Auditing:** Implement periodic key rotation and maintain logs to detect anomalies and perform security audits, especially after device compromise.

b) Frame Counter Integrity and Resynchronization

- **Server-Side Alerts for FCnt Jumps:** Trigger alerts when abnormally high jumps in **FCnt** are detected, especially outside known device behavior patterns as shown in Figure 4.13.

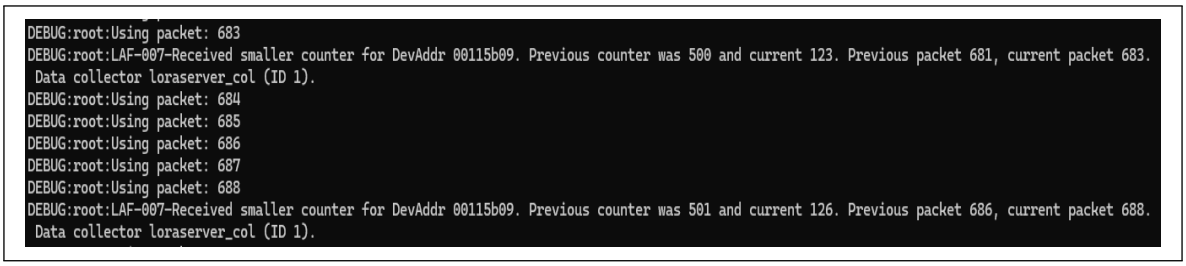


Figure 4.13: Desynchronization of Chirpstack due to the attack

c) Anomaly Detection and Traffic Monitoring

- **Real-Time Analytics and Monitoring:** Use tools like Grafana and Prometheus to visualize trends and detect inconsistencies in GPS data, signal strength, or packet frequency.

- **Heuristic and Signature-Based Attack Detection:** Employ intrusion detection systems (IDS) tailored for LoRaWAN to identify suspicious traffic patterns, such as repeated replays or forged metadata.
- **Rate Limiting and Gateway Whitelisting:** Apply uplink rate limiting and accept traffic only from known, authenticated gateways to prevent rogue packet injections.

d) Application Layer Validation

- **Sanity Checks on Sensor Data:** The application layer should validate payload contents (e.g., GPS bounds, timestamps) to flag impossible or abnormal values.
- **Use Signed Payloads (Optional):** Consider implementing application-layer digital signatures to further authenticate the origin of the payload, even if network-level encryption is compromised.

Conclusion

The vulnerabilities demonstrated through our crafted desynchronization attack stem not from protocol flaws alone, but from insecure deployments and lack of layered defenses. By applying these mitigations spanning key management, monitoring, protocol hardening, and application logic validation LoRaWAN networks can significantly increase resilience against both passive and active adversaries.

GENERAL CONCLUSION

This project has provided a comprehensive security analysis of LoRaWAN networks through the development of the LoRaXploit framework and practical attack simulations. By successfully demonstrating real-world vulnerabilities like desynchronization attacks and cryptographic key extraction, we have highlighted critical weaknesses in current LoRaWAN implementations. Our testbed environment, integrating both attack simulation tools and defensive monitoring systems, has proven particularly effective in evaluating network resilience. The results confirm that while LoRaWAN's architecture provides fundamental security features, real-world deployments remain vulnerable to sophisticated attacks when proper countermeasures are not implemented. These findings underscore the importance of robust key management, frame counter protection, and continuous network monitoring in real-world IoT deployments.

Future Perspectives Looking ahead, this work opens several promising directions for both research and practical implementation. Immediate priorities include the development of hardware-based security modules for end devices and the integration of dynamic key rotation protocols. The implementation of machine learning-driven anomaly detection systems could significantly improve threat identification, while updates to LoRaWAN specifications should address the identified vulnerabilities. Future work should also explore the security implications of newer protocol versions and their interaction with emerging IoT architectures. As IoT adoption expands, such security advancements will become increasingly vital for safeguarding critical infrastructure and ensuring user privacy. This study lays important groundwork for future efforts to strengthen LoRaWAN security across academic and industrial domains.