A cloud based key rolling technique for mitigating the man-in-the-middle attacks in LoRaWAN infrastructures

by

NIKOLAOS ASTYRAKAKIS

B.Eng. in Network Engineering,
Hellenic Mediterranean University, 2019

A THESIS

submitted in partial fulfilment of the requirements for the degree

MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2022

Approved by:

Evangelos Pallis
Evangelos Markakis

Μια τεχνική νέφους συνεχούς εναλλαγής κλειδιών για την εξάλειψη επιθέσεων Man-In-The-Middle σε LoRaWAN υποδομές.

από

Νικόλαος Αστυρακάκης

Πτυχίο Μηχανικού Δικτύων Πληροφορικής
Ελληνικό Μεσογειακό Πανεπιστήμιο, 2019

Η μεταπτυχιακή εργασία

υποβάλλετε σε μερική εκπλήρωση των απαιτήσεων για το πτυχίο

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ «ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ»

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

2022

Εγκρίθηκε από:

Ευάγγελος Πάλλης
Ευάγγελος Μαρκάκης

Abstract

Recently, the Internet of Things (IoT) has emerged to a trending technology. IoT devices may capture, analyse, and process data before transferring it to application servers or sharing it through wired or wireless networks, across great distances. The LoRaWAN network is well-suited for such situations because it employs Low Power (LP) and Long Range (LoRa) protocols that reduce device energy consumption while maximising communication range. Before data transmission, all IoT devices on a LoRaWAN network must authenticate through a gateway to the backhaul, following an initial Join Procedure based on the submission of a Join Request. A Join Request includes two identifiers (AppEUI, DevEUI), a random value (DevNonce), a Media Access Control (MHDR) header, and a Message Integrity Code (MIC). The MIC value is the encryption result of either one of the aforementioned values or all of them together with an AppKey that is also stored in the backhaul for device authentication upon Join Request retrieval. Malicious actors acting as Man-In-the-Middle (MITM) can interfere in the channel and reverse engineer the AppKey; thereby, they may initiate a Join Request, which is misinterpreted as coming from a legitimate device, and eventually join the communication channel. This thesis presents a prototype technique that allows IoT devices to re-authenticate before every message exchange session, by constantly regenerating the AppKey. The suggested method uses a similar key rolling technique as the one used in automobile central locking systems, enhances the network's resistance against MITM attacks, and is developed as an optimized and scalable microservice for various LoRaWAN installations and versions.

Περίληψη

Πρόσφατα, το Internet of Things (IoT) έχει αναδειχθεί σε μια μοντέρνα τεχνολογία. Οι συσκευές IoT εσωτερικού και εξωτερικού χώρου ενδέχεται να καταγράφουν, να αναλύουν και να επεξεργάζονται δεδομένα πριν τα μεταφέρουν σε διακομιστές εφαρμογών ή τα μοιραστούν μέσω ενσύρματων ή ασύρματων δικτύων, σε μεγάλες αποστάσεις. Το δίκτυο LoRaWAN είναι κατάλληλο για τέτοιες καταστάσεις, επειδή χρησιμοποιεί πρωτόκολλα χαμηλής ισχύος (LP) και μεγάλης εμβέλειας (LoRa) που μειώνουν την κατανάλωση ενέργειας της συσκευής ενώ μεγιστοποιούν το εύρος επικοινωνίας. Πριν από τη μετάδοση δεδομένων, όλες οι συσκευές IoT σε ένα δίκτυο LoRaWAN πρέπει να πραγματοποιήσουν έλεγχο ταυτότητας μέσω μιας πύλης στο backhaul, ακολουθώντας μια αρχική Διαδικασία Συμμετοχής που βασίζεται στην υποβολή Αίτησης Συμμετοχής. Ένα αίτημα συμμετοχής περιλαμβάνει δύο αναγνωριστικά (AppEUI και DevEUI), μια τιμή τυχαίου μηνύματος (DevNonce), μια επικεφαλίδα (MHDR), και έναν κωδικό ακεραιότητας μηνύματος (MIC). Η τιμή MIC είναι το αποτέλεσμα της κρυπτογράφησης είτε μίας από τις προαναφερθείσες τιμές ή όλων των τιμών μαζί με ένα AppKey που αποθηκεύεται επίσης στο backhaul για έλεγχο ταυτότητας συσκευής κατά την ανάκτηση του αιτήματος συμμετοχής. Κακόβουλοι ηθοποιοί που ενεργούν ως Man-In-the-Middle (MITM) μπορούν να παρέμβουν στο κανάλι και να δημιουργήσουν αντίστροφη μηχανική του AppKey. Ως εκ τούτου, μπορούν να ξεκινήσουν ένα Join Request, το οποίο παρερμηνεύεται ότι προέρχεται από μια νόμιμη συσκευή, και τελικά να ενταχθούν στο κανάλι επικοινωνίας. Αυτή η εργασία παρουσιάζει έναν πρωτότυπο μηχανισμό που επιτρέπει στις συσκευές IoT να ελέγχουν εκ νέου την ταυτότητα πριν από κάθε συνεδρία ανταλλαγής μηνυμάτων, αναδημιουργώντας συνεχώς το AppKey. Η προτεινόμενη μέθοδος χρησιμοποιεί παρόμοια τεχνική κυλιόμενου κλειδιού με αυτή που χρησιμοποιείται στα συστήματα κεντρικού κλειδώματος αυτοκινήτων, ενισχύει την αντίσταση του δικτύου έναντι επιθέσεων MITM και αναπτύσσεται ως βελτιστοποιημένη και κλιμακούμενη μικρουπηρεσία για διάφορες εγκαταστάσεις και εκδόσεις LoRaWAN.

Table of Contents

Abstract 3

Περίληψη 4

Table of Contents 5

Acknowledgements 8

Chapter 1 Introduction 9

Chapter 2 Literature Review 14

Chapter 3 Technology enablers 17

LoRaWAN Networks 17

Chirpstack Framework 18

PyCom Devices (FiPy and PySense Shield) 20

Lorix One (Gateway) 21

Docker and Docker Compose 21

Node.JS Server 22

RestAPI Interface 22

Gitlab 23

Postman 23

PyMakr Extension and Visual Studio Code 23

Chapter 4 System Architecture and Conceptual Diagram 24

Chapter 5 Implementation 27

Description of Implementation 28

Sequence Diagram 31

Flow Chart – Diagram 33

Chapter 6 Evaluation 34

Chapter 7 Conclusion 39

References 40

Appendixes 42

APPENDIX I: End device code 42

- Device folder structure: 42
- Device Code (Python) (main.py): 42

APPENDIX II: Rolling key module code 46

- Device folder structure: 46
- Main Rolling Key module code (main.js): 46
- Device Data Json Analyzer Extracted from Chirpstack (Evaluation): 54
- Rolling key “Delay” Experiment data analyser (Evaluation): 55

APPENDIX III: Instructions for deploying and setup of Chirpstack framework 57

APPENDIX IV: Instructions for flashing / programming gateway 59

APPENDIX V: Instructions for installing devices to Chirpstack framework based on DevEUI 62

List of Figures

Figure 1 Messages, values, keys, and information generated and exchanged between an extreme edge device and the backhaul during the Join Procedure in LoRaWAN networks operating in OTAA mode. 11
Figure 2 LoRaWAN network architecture 17
Figure 3 Chirpstack User Interface 18
Figure 4 Chirpstack RestAPI 19
Figure 5 PyCom Microcontroller 20
Figure 6 PySense Shield 20
Figure 7 Lorix One 21
Figure 8 RestAPI In Action 22
Figure 9 Architecture of LoRaWAN network with Key Rolling Module 24
Figure 10 Conceptual Diagram 25
Figure 11 LoRaWAN Infrastructure and testbed devices 27
Figure 12 Chirpstack User Interface 28
Figure 13 Sequence Diagram 32
Figure 14 Flow Diagram 33
Figure 15 Evaluation testbed 34
Figure 16 Delay Experiment Results 36
Figure 17 Average packet per minute (ppm) 37
Figure 18 Accepted and Rejected Packets 38

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Evangelos Pallis, Director and Head of PASIPHAE Lab, Professor at the Department of Electrical and Computer Engineering, Hellenic Mediterranean University (currently acting as a Professor at the Department of Industrial Design and Production Engineering, University of West Attica), for his guidance, encouragement, remarks, and assistance in completing this master's thesis in its entirety.

Second, I'd like to thank my second supervisor, Dr. Evangelos Markakis, Principal Investigator at Hellenic Mediterranean University Research Center, for his valuable comments and motivation in completing this thesis on time.

Finally, I'd like to especially thanks my colleague Mrs. Dimitra Papatsaroucha for hours of assistance and all my colleagues from PASIPHAE lab for their valuable comments and support. Additionally, a heartfelt thanks to my mother, father, girlfriend and sisters for their unconditional support during my studies.

That’s it, I’m done!

Chapter 1 Introduction

In the newest technology era, the Internet of Things (IoT)[1] is constantly expanding, enabling the use of smart devices in most communities and areas, from smart homes to smart cities, fields, and more. These devices are designed for indoor and outdoor use and they can transmit data across distances of several kilometres. Meanwhile, the use of specialised protocols for regulating data flows and device sleep intervals allows some of these devices to operate with reduced battery consumption, depending on the use case and configuration.

Low Power Wide Area Networks (LPWAN), such as LoRaWAN[2], SigFox[3], and NB-IoT[4] (GSM based), are specialized for long-distance data transmissions. These networks are usually employed in agriculture, manufacturing, logistics, and a variety of other outdoor and indoor applications for sensor data transmission, as shown in [1]. Devices utilising such networks can be used to send sensitive or personal data over the air. However, the research community is concerned about the security mechanisms and procedures that are currently employed in these networks, due to the fact that the air is a fairly insecure medium for data transmissions.

According to the current LoRaWAN network security approach and the version 1.0.2 of the LoRaWAN specification[5], each edge IoT device must be activated and personalised in order to participate in a LoRaWAN network. Activating an extreme edge device can be accomplished either by following the Over-The-Air-Activation (OTAA) security verification process or through Activation by Personalization (ABP). Following the OTAA mode, an extreme edge device sends unencrypted Join Requests to the backhaul during the initial device authentication procedure, which is called Join Procedure. Activating an extreme edge device through personalization combines the two steps of extreme edge device personalization and activation and bypasses the Join procedure. Attacks that are known as "impersonation" attacks follow standard principles of capturing, tampering, and retransmitting data in a network while pretending to be someone else. An impersonation attack affects data accuracy, which is vital in critical infrastructures, which rely

[1] https://en.wikipedia.org/wiki/Internet_of_things
[2] https://lora-alliance.org/about-lorawan/
[3] https://www.sigfox.com/en
[4] https://en.wikipedia.org/wiki/Narrowband_IoT
[5] https://lora-alliance.org/resource_hub/lorawan-specification-v1-0-2/

on sensor data to complete numerous consecutive activities. Storing tampered data in the backhaul without validation and supervision may result in a disaster.

Join Requests comprise, among others, two identifiers (AppEUI and DevEUI), a random message value (DevNonce), a Media Access Control (MHDR) header, and a Message Integrity Code (MIC) value. AppEUI and DevEUI are utilised for server and device identification, respectively. The AppEUI value is a global ID that is utilised to distinguish the entity that can process the Join Request among several available Join servers in the backhaul. The DevEUI value is a 64-bit globally unique identifier of each extreme edge IoT device and is provided either by the manufacturer or the owner of the extreme edge device. The DevNonce value is a random generated number that is utilised for preventing replay attacks. Using this number, the backhaul keeps track of the DevNonce values that an extreme edge device has already used in its Join Requests and rejects future requests that contain a DevNonce value that has already be stored in the backhaul. The MAC header defines the message type and major version of the frame format used by the LoRaWAN layer specification to encode the frame. The MIC value is the result of the encryption of either the AppEUI, the DevEUI, the DevNonce, or all of them together with a key called AppKey. The AppKey is randomly generated in the backhaul for each extreme edge device that needs to communicate with the cloud servers and is manually inserted into each extreme edge device. When a Join Request reaches the backhaul, the AppKey is used for decrypting the MIC value and, therefore, authenticating the extreme edge device that sent the Join Request. Then, the AppKey is utilised by the backhaul for generating the AppSKey and the NwkSKey through the encryption of either the DevNonce or a value called AppNonce. The AppNonce is a randomly generated value or a predefined unique ID in the backhaul, which is transmitted from the backhaul to the extreme edge device inside an encrypted Join Accept message that concludes the Join Procedure. The AppNonce is then used by the extreme edge device alongside the AppKey for generating the same AppSKey and the NwkSKey as the ones stored in the backhaul. Following, these session keys are utilised by both the extreme edge device and the backhaul for encryption and verification of network communication and application data. The contents of the Join Accept message are outside the scope of this study as these messages are encrypted before transmission, in contrast to Join Request messages. All messages, values, keys, and information generated and exchanged between an extreme edge device and the backhaul during the Join Procedure are depicted in Figure 1.

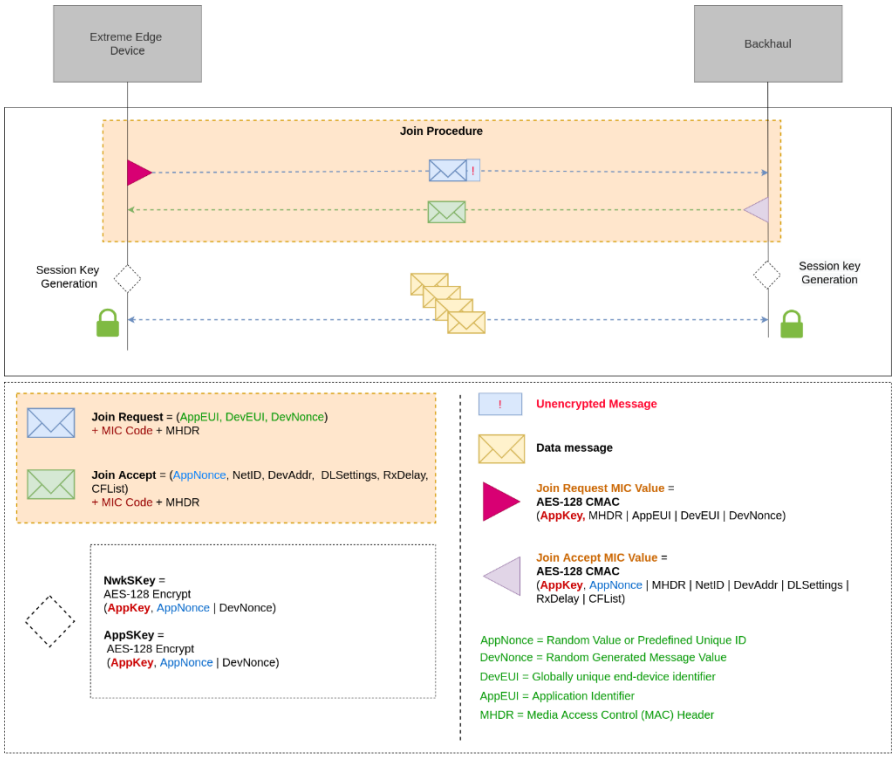


Figure 1 Messages, values, keys, and information generated and exchanged between an extreme edge device and the backhaul during the Join Procedure in LoRaWAN networks operating in OTAA mode.

Although, all LoRaWAN networks follow the specifications established by the LoRa Alliance and are described as secure, the security of these networks is argued across relevant literature and several open issues have been identified. For example, in [2] is described that LoRaWAN networks operating in OTAA mode lack security measures during the Join Procedure while Join Requests are transmitted unencrypted over the air; thus, devices that use LoRaWAN networks over long distances are vulnerable to eavesdropping and MITM attacks, such as replay attacks[6] shown in [3]. An attacker may intervene in the communication and try to derive the AppKey, through reverse engineering of the MIC value, and eventually manipulate the data by mimicking the behaviour of the victim device - an attack known as impersonation attack. Although the MIC value transmitted over LoRaWAN networks is produced by the AES-CMAC algorithm, which uses AES-128 blocks for encryption that are difficult to be cracked, Isa H. et al in [4] showcased several algorithms and techniques that may reduce the processing power needed in order to decrypt faster any AES-128

[6] https://en.wikipedia.org/wiki/Replay_attack

messages. Moreover, the AppEUI and DevEUI identifiers, the DevNonce value, and the MHDR header are transmitted unencrypted over the air. They also remain the same for each Join Request a specific extreme edge device sends, alongside the AppKey that is also a static, one-time generated key. What is more, research has revealed that the DevNonce value can be easily intercepted through a brute force attack [3]. As a result, there is an increased probability that an attacker may access this unencrypted information, decrypt the MIC value, derive the AppKey, and send a Join Request that is misinterpreted by the backhaul as coming from a legitimate device. Thereby, a malicious actor may manipulate and store tampered data in the backhaul which may lead to disaster, particularly in critical infrastructures, such as healthcare or pollution monitoring, as depicted in [5].

This thesis presents a novel technique for enhancing the security level of LoRaWAN networks by securing the authentication process, described in the LoRaWAN v1.0 specification document[7] , and specifically the Join Procedure of the OTAA technique. A key rolling technique, also known as rolling code, similar to the one used in car central locking systems, is proposed to be modified and integrated with LoRaWAN networks for providing an upgraded security technique. Through the use of the key rolling technique, the static AppKey utilised for the AES-128 encryption of AppEUI, DevEUI, DevNonce, or MHDR into the MIC value during the Join Procedure is replaced with a dynamic key that changes in every message exchange session. The constant and continuous change of the AppKey, alongside the AES-128 encryption, make it highly challenging for attackers to intercept and tamper with the communication channel. That is because an adversary would require a great amount of processing power in order to capture and reverse engineer the MIC value for deriving the AppKey, before the AppKey changes again. The proposed implementation was tested in a real-world scenario with actual PyCom FiPy Micro-controllers (extreme edge devices) acting as LoRaWAN clients, a Lorix One (edge device) acting as the gateway, and a private self-hosted Chirpstack framework serving as the backhaul of the LoRaWAN network. The results of extensive testing showed that this novel technique improves the security of LoRaWAN networks and prevents malicious users from manipulating data and compromising critical infrastructures.

The rest of this thesis is structured as follows: Chapter 2, where state of the art endeavours and techniques to secure the Join procedure of the OTAA mode in LoRaWAN and LPWAN networks are described; Chapter 3, where the technologies utilized to construct the proposed system are

[7] https://lora-alliance.org/wp-content/uploads/2020/11/lorawan1_0_2-20161012_1398_1.pdf

depicted; Chapter 4, where the proposed system architecture is presented alongside a description about each system component; Chapter 5, which showcases the implementation of the proposed key rolling technique; Chapter 6, which depicts the results after the evaluation of the proposed system in a real world environment with a physical attacker and the key rolling technique in function; Chapter 7, where concluding thoughts and proposed future steps are presented.

Chapter 2 Literature Review

As the number of IoT networks has increased in recent years, so has the research community's interest in these networks, while several research endeavours have revealed the unresolved challenges of such networks. LoRaWAN networks can transfer important data in an industrial infrastructure to enable automated sequential activities, or they can transport personal data in a common residential infrastructure. Researchers of the field have detected several LoRaWAN open issues, as well as a number of ways to mitigate those vulnerabilities. This section reviews state-of-the-art approaches towards enhancing the security level of LoRaWAN networks.

Kim J. and Song J. in [6] proposed an authentication technique that "replaces" the DevNonce number with two similar Join Procedures. The first one, named Initial Join Procedure, is the same with the Join Procedure defined by the LoRa Alliance in the specification regarding device authentication. The second procedure is named Non-Initial Join Procedure and it was developed by the authors for enhancing the security of LoRaWAN networks by performing a secondary custom device authentication. However, the keys of the Initial Join procedure remain static and unencrypted while they are transmitted over the air, thus, there is a high possibility that the whole network is still at risk.

The vulnerability of the Join Procedure was also pointed out by SeungJae Na, et al. in [2], who proposed a technique for masking the DevNonce and MIC values of the Join Request. Masking is the procedure of masquerading a sequence of characters or numbers by keeping the same format and using a string or number as a masking token alongside a logic operation, such as XOR, AND, etc., which are often used in cryptography. The proposed method extracts the initial masking token from the AppKey and uses it for masking the DevNonce and MIC values of the Join Request. Then, the NwkSKey of each data transmission session is used for masking the DevNonce and MIC values of the next transmission session. The entire system is reported to be safe, yet this assumption is primarily based on the knowledge, skills, and computing power of the attacker as well as on the results of the recon that is performed for gathering information about the network prior to the attack. An attacker may capture a Join Request message, decrypt the AES-128 generated MIC value to extract the AppKey, and recreate a legitimate Join Request. Then, after joining the network, the attacker can access the NwkSKey of each data transmission session and use it to mask the DevNonce and MIC values of the next transmission session. Thus, the system remains

vulnerable as malicious users can recreate this sequence of actions and exploit the LoRaWAN network.

On a similar note, Thomas J. et al. in [7] presented a modified masking algorithm to prevent deriving the AppKey by decrypting the MIC value of the Join Request. The authors proposed a modified cryptography counter mode algorithm that performs differently than the default block cipher mode (CTR) used in AES-128 encryption. The proposed counter introduces multiple XOR functions for masking the Join Request before transmission, in contrast to the default procedure which uses a single XOR function. The authors proved that the proposed approach increases the difficulty of interfering in a communication channel and modifying the transmitted data; however, this remains analogous to the knowledge, skills, and computing power of the attacker. Additionally, a performance analysis regarding power consumption and processing power impact on extreme edge nodes (clients) of the proposed implementation is yet to be conducted.

Another method for securing the Join Procedure was proposed by Sung, W. J. et al in [8]. This method relies on the backhaul and utilises the Received Signal Strength Indicator (RSSI) that shows how far a device may be depending on its signal strength. Moreover, a handshaking technique alongside proprietary messages are used to ensure integrity of messages and prevent replay attacks. However, this solution lacks the validation of the RSSI value around a gateway; therefore, an attacker being in the same distance as a legitimate device and having the same RSSI may be misinterpreted as legitimate device.

Furthermore, Xia, Z. et al. in [9] proposed a Key Distribution Server (KDS) that introduces a decentralised key provision infrastructure for securing the Join Procedure. This infrastructure, which generates session keys for the extreme edge LoRaWAN devices and verifies keys received from the backhaul, partially resolved the Join Procedure vulnerability; however, at the expense of increased network complexity, packet size, and latency.

Finally, Lin, J. et al. in [10] and Danish, S. M. et al. in [11] presented a technique for securing LoRaWAN networks and ensuring packet integrity using the blockchain concept. In [10] the authors proposed a blockchain schema for constructing a secure LoRaWAN network, using the blockchain architecture in the backhaul, and ensuring data integrity for all received messages. Furthermore, in [11] a blockchain two-factor authentication module was proposed, which can be enabled on the gateway of any LoRaWAN network. In the latter case, the gateway is responsible for verifying the integrity of received packets with the Ethereum blockchain before transporting

the packets to the backhaul. Although these methods are effective for safeguarding LoRaWAN networks, the authors noted that physical attacks might compromise the blockchain keys and, as a result, the entire security mechanism upon which LoRaWAN networks rely.

Chapter 3 Technology enablers

This chapter describes the cutting-edge technologies used in this thesis to tailor the rolling key technique with LoRaWAN networks as efficiently as possible.

LoRaWAN Networks

LoRaWAN[8] is a network architecture/specification for LoRa networks that is based on LPWAN principles, such as range, geographical coverage, transmission latency, and delivery of data over prolonged distances, while minimising device battery consumption (in case devices are battery powered). It is an open source network specification developed by the LoRa Alliance[9] and is commonly used in Internet of Things (IoT) extreme edge devices, such as PyCom, Arduino, and others.

The LoRaWAN network architecture is depicted in Figure 2. This figure depicts the extreme edge (device – on the left), the edge devices (gateway – on the middle) and the backhaul of the network (circle – on the right). The backhaul modules (circle) are: (1) the application servers who are responsible for storing and interacting with the received data from devices, (2) the network servers who are responsible for the gateway network communication, routing of data and authentication of devices and (3) in the latest versions of LoRaWAN, there is a join server that is responsible for the authentication of devices, separated from the network server. All components described above are depicted, from the left to the right, respectively.

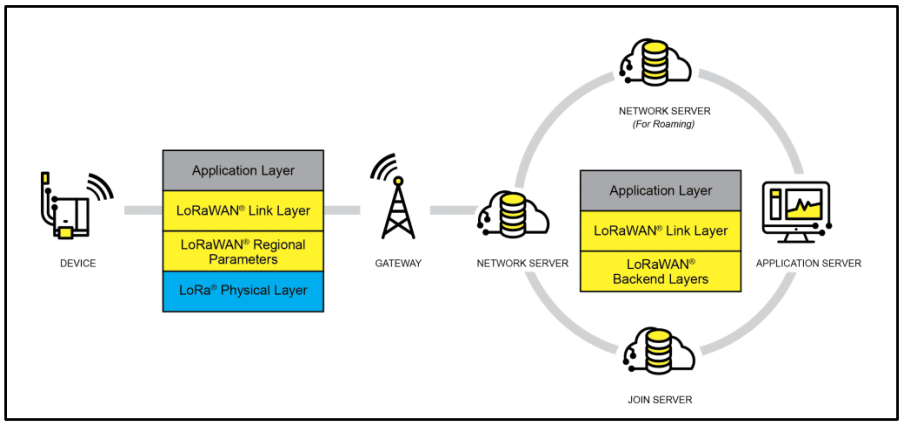


Figure 2 LoRaWAN network architecture

[8] LoRaWAN: https://lora-alliance.org/about-lorawan/
[9] LoRa Alliance: https://lora-alliance.org

Chirpstack Framework

This thesis required a LoRaWAN framework to store, monitor and manage data send from the extreme edge devices (LoRaWAN devices), to the backhaul. Such frameworks are: The Things Network (TTN)[10], the Chirpstack framework[11], LORIOT[12] and etc. This thesis elaborated the Chirpstack framework, which provides an intuitive user interface as depicted in Figure 3. The Chirpstack is a cloud framework used to deploy a network and an application server, as backhaul because it provided all the necessary interfaces to communicate such as RestAPI, WebSockets and etc. A Chirpstack RestAPI preview is depicted in Figure 4. The application server of this framework is in charge of storing the data transmitted by IoT devices, while the network server is in charge of managing the transportation of such data from the device to the application server, through the gateway. Furthermore, the Chirpstack network server is in charge of managing the entire LoRaWAN network packets.

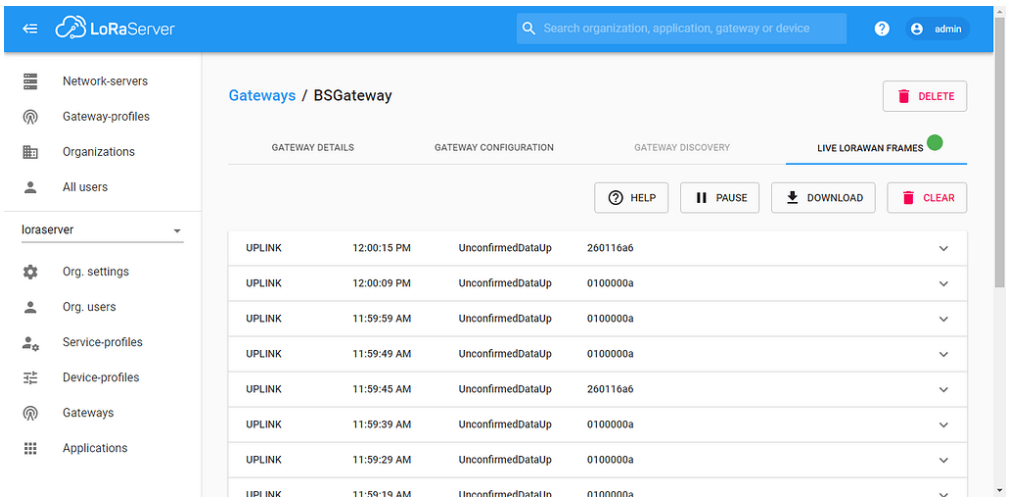


Figure 3 Chirpstack User Interface

[10] https://www.thethingsnetwork.org/
[11] https://www.chirpstack.io/
[12] https://www.loriot.io/

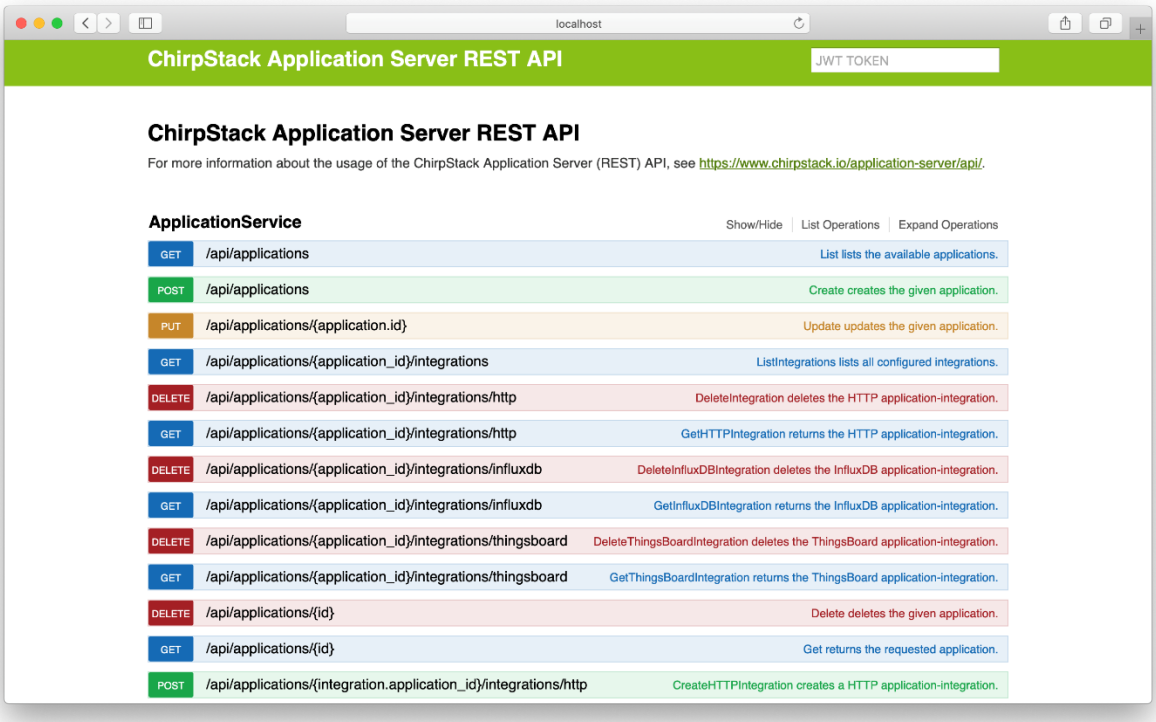


Figure 4 Chirpstack RestAPI

PyCom Devices (FiPy and PySense Shield)

PyCom is a manufacturer of microcontrollers (e.g. FiPy, depicted in Figure 5) and sensors (e.g. PySense, depicted in Figure 6) used in the implementation of wireless networks and automations. PyCom provides a simple interface for programming the microcontrollers with Python programming language and it outstands the competition in terms of compatibility and modularity. In this thesis, the FiPy microcontroller, from PyCom, was utilised as it provided all the available libraries and NAND Flash memory, required for storing any value. Along with the FiPy microcontroller, this thesis utilized the PySense sensors' shield to measure the temperature, humidity, light, and other ambient information around the IoT Device (FiPy). The FiPy device was used in this thesis to set up a LoRaWAN client connection to the private LoRaWAN network – backhaul - which was set up on premise and it was programmed to send the aforementioned metrics to the Chirpstack Framework (backhaul) described above.

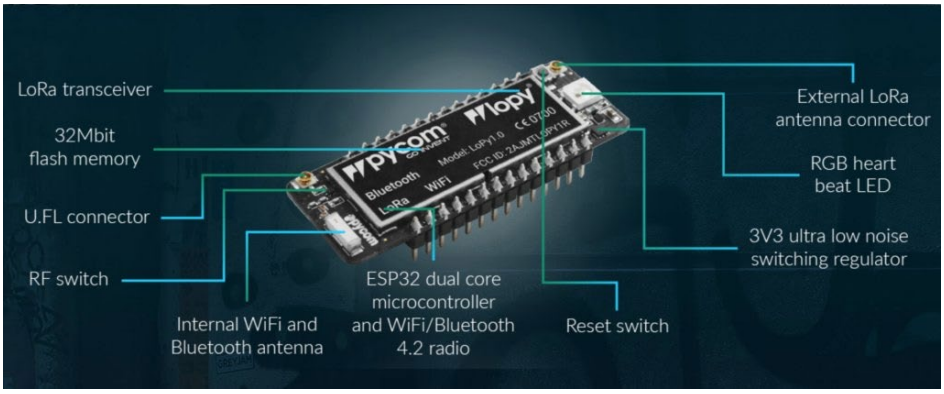


Figure 5 PyCom Microcontroller



Figure 6 PySense Shield

Lorix One (Gateway)

Various gateways are available in Commercial off-the-shelf (COTS) products. The Lorix One[13], depicted in Figure 7 is selected for this thesis system, because it uses an embedded (ARM) microcontroller, a Linux operating system (OS) and an antenna for receiving or transmitting data from/to extreme edge devices in half-duplex mode. Lorix One acts as a Gateway and it was used in this thesis' LoRaWAN network, for receiving and transferring sensor data to the backhaul servers.



Figure 7 Lorix One

Docker and Docker Compose

Containers can be built and run with various frameworks such as Docker , Containerd and etc. Most popular choice for developers is Docker and a tool named Docker Compose. Docker allows the containerization of applications with a Dockerfile that describes the process of building and running the application and docker-compose allows the execution of containerized applications and the provision of network and storage. Containerized applications and microservices are becoming more prevalent in the current era. Such kinds of applications provide fast and easy scalability, management, redundancy and resiliency.

[13] Lorix One: https://www.lorixone.io/en

Node.JS Server

Building the key rolling mechanism required a backend to run in the containerized environment described above. The Node.js[14] server was selected for this purpose. Node.js is an application server that enables the execution of Javascript code in the back-end side and it runs natively in a containerized environment. Node.js is fairly easy to use and write code for it, as it has a huge community support, libraries and solutions to issues. Additionally, it allows easier access to RestAPIs and communication interfaces using minimal code and libraries.

RestAPI Interface

RestAPI was selected for connecting this thesis system to the Chirpstack framework, because of limited complexity in comparison to GRPC protocol that Chirpstack offers. The Chirpstack RestAPI depicted in Figure 8 can be found in during execution of Chirpstack deployments, on server's URL (e.g. https://example.com/api/ or https://192.168.1.123/api).

The RestAPI[15] is an interface, as shown in Figure 8, exposed to a network, by an application, which allows various HTTP requests to the application. Utilizing a RestAPI interface, devices/actors can save data to a database, authenticate or perform various actions on a system.

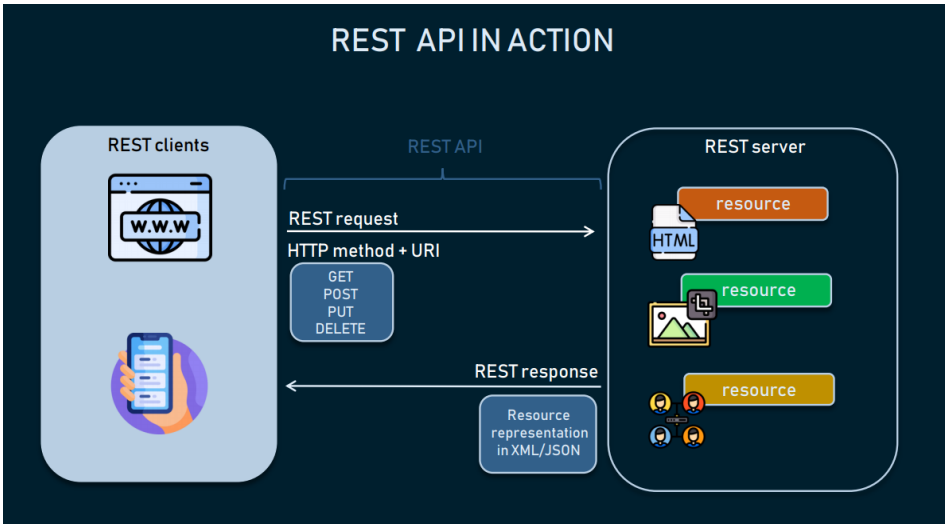


Figure 8 RestAPI In Action

[14] https://nodejs.org/en

[15] https://www.redhat.com/en/topics/api/what-is-a-rest-api

Gitlab

Gitlab[16] is a code repository, that primarily stores code, files and containerized application images. It was used for agile development of the rolling key microservice as well as for providing documentation and instructions regarding the replication of the LoRaWAN network and the device firmware flashing that was used in this thesis. The Gitlab was provided by the Pasiphae Lab of the University.

Postman

Postman[17] is a tool that allows users to perform various actions by utilising the HTTP and HTTPS protocols. It has an intuitive user interface in comparison to other applications such as curl, wGet, etc that use CLI for performing requests. Postman is fairly easy to use in order to perform various kinds of requests such as GET, POST, PUT, HEAD, etc. This tool assisted in discovering and performing requests to test the proof of concept of Rolling Keys with the Chirpstack framework via the RestAPI interface, described above, before implementing the main codes and microservices that automate the whole process of this thesis.

PyMakr Extension and Visual Studio Code

PyMakr[18] is a tool - extension - that can be installed in the Visual Studio Code IDE. It is used to flash the custom firmware to the PyCom devices, and, additionally, preview the serial monitor of these devices, which is connected through a Universal Serial Bus (USB) interface.

[16] GITLAB: https://about.gitlab.com/
[17] Postman: https://www.postman.com/
[18] Pymakr: https://pycom.io/products/supported-networks/pymakr/

Chapter 4 System Architecture and Conceptual Diagram

This section presents the components of the system architecture upon which the proposed security technique was implemented, namely: the backhaul, the edge device, and the extreme edge devices (LoRaWAN clients).

The system architecture diagram in Figure 9 illustrates the infrastructure components of the LoRaWAN network, the key rolling module, and the interconnection protocols used by the following layers: the edge devices (gateways), the extreme edge devices (LoRaWAN clients), and the backhaul (network/application servers). Additionally, a conceptual diagram in Figure 10 depicts all the components of the proposed system as well as the interfaces utilised to protect the LoRaWAN v1.0 networks.

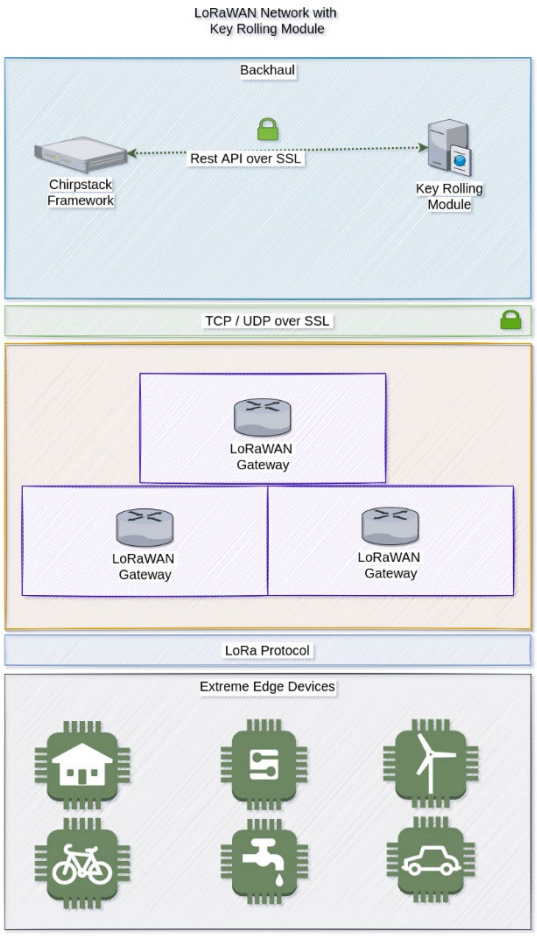


Figure 9 Architecture of LoRaWAN network with Key Rolling Module

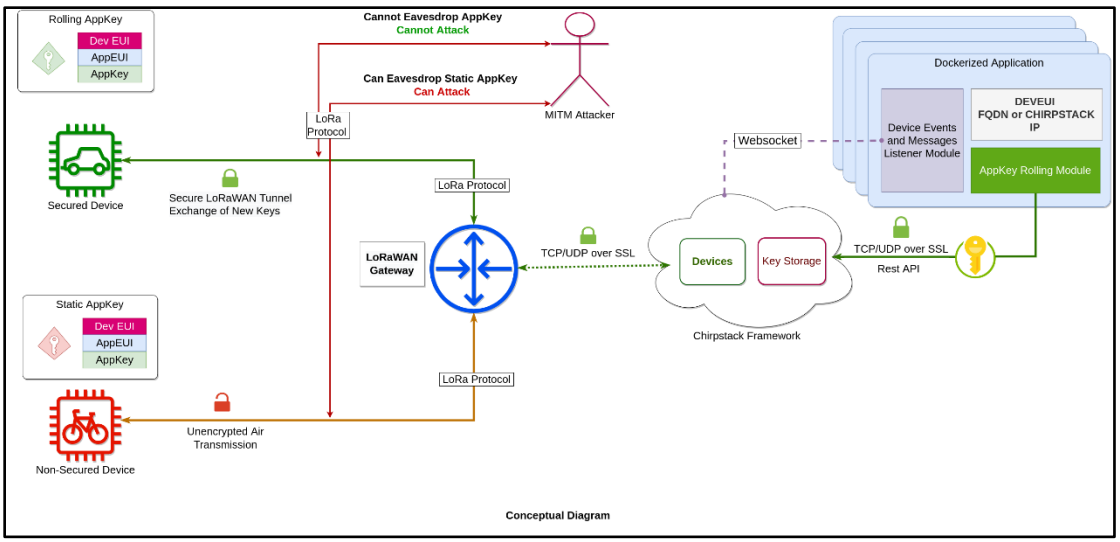


Figure 10 Conceptual Diagram

On the left side of the conceptual diagram one can see an example of two extreme edge LoRa devices: the device at the top is secured with the proposed key rolling technique and has a new rolled AppKey in every communication; the device at the bottom is not secured and performs connections to the privately deployed Chirpstack framework using a static AppKey. In the centre of the conceptual diagram, one may view the gateway, which oversees collecting all communications and forwarding them to the appropriate backhaul, and a potential attacker acting as a Man-In-The-Middle, attempting to gain access to both secured and unsecured devices. Furthermore, on the right side of the figure one may notice the Chirpstack framework, with which the gateway connects via a secure TCP connection over SSL certificates, and the Cloud native Application (microservice), which listens to messages arriving at the Chirpstack framework and triggers specific actions.

The aforementioned microservice utilises a WebSocket connection to monitor a certain device, according to its DevEUI, while it also collaborates with the key rolling module to re-roll the AppKey on every new communication of this specific device. Every device is controlled by a certain, dynamic allocated microservice defined in a finite list. This list includes the identifiers of a device (i.e., AppEUI and DevEUI) and the Domain name or IP Address of the Chirpstack Framework. The microservice inputs (environment variables) are: the DevEUI of the device to be monitored; the current AppKey, through which the device authenticates to the Chirpstack framework; the Domain Name or IP Address of the Chirpstack framework to which the application

data is logged. The microservice can be deployed in the same local area network (LAN) as the Chirpstack framework or in a different network if it has access to the Chirpstack framework. It can also communicate with a Chirpstack framework through the Internet in case the microservice is deployed on a cloud infrastructure. Regarding the last option, the Chirpstack framework must have trusted certificates (SSL) and a fully qualified domain name (FQDN). Finally, as it is shown in the conceptual diagram, the cloud native application transports each new rolled AppKey both to the backhaul and the extreme edge LoRa device. This transmission is conducted through the secured Rest API interface that is exposed by the Chirpstack framework and the gateway and contains the respective downlink message for each entity.

Chapter 5 Implementation

The first section of this chapter describes the implementation of the key rolling module and the integration of the key rolling technique with LoRaWAN version 1.0 networks. In this proposal, a few assumptions were made prior to the implementation: the AppEUI that was utilized in all extreme edge devices is set as "0000000000000000", because the Chirpstack framework, in particular, does not require an AppEUI and ignores its value in case it is set. Additionally, the version 1.0.2 LoRaWAN specification was used was for this implementation because the utilised devices do not support higher versions of LoRaWAN networks, such as 1.1.

Following, a sequence diagram and a flow chart are depicted for better understanding of the actions taken by the key rolling module. The testbed and LoRaWAN infrastructure used for the proposed implementation are illustrated in Figure 11. The Chirpstack interface for manipulating any stored data and registered devices is showcased in Figure 12.

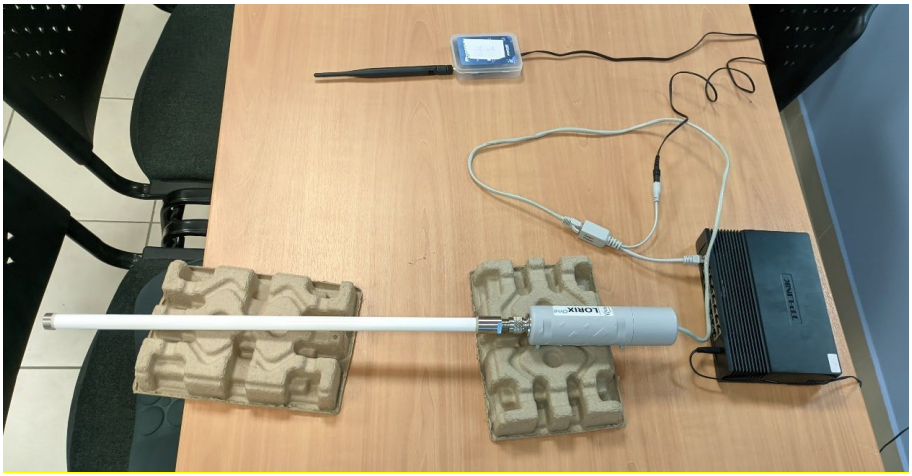


Figure 11 LoRaWAN Infrastructure and testbed devices

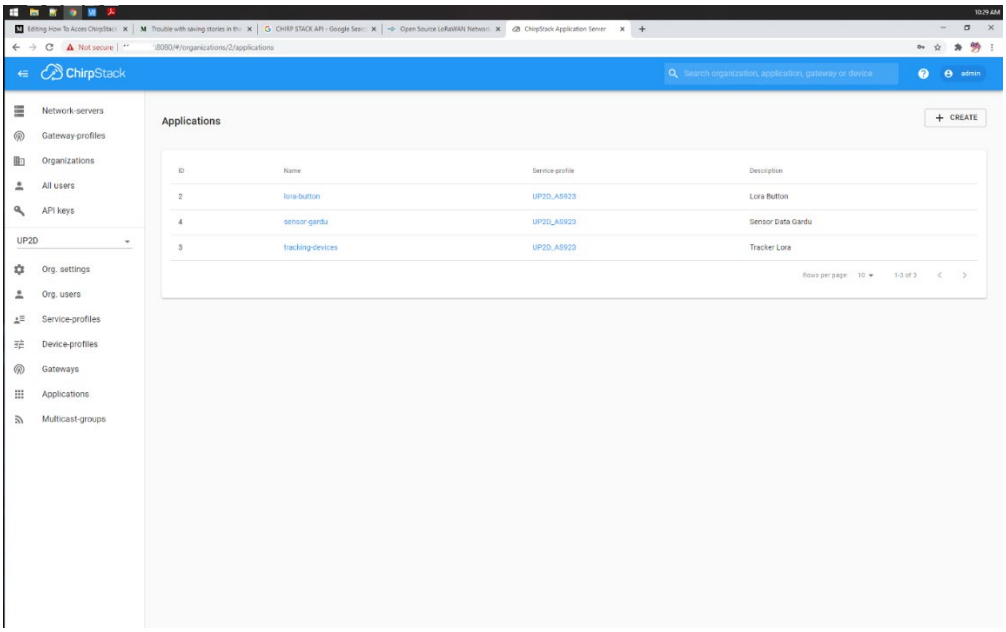


Figure 12 Chirpstack User Interface

Description of Implementation

The goal of this thesis is to demonstrate a rolling key concept in LoRaWAN networks in order to improve the security of LoRaWAN networks. This thesis uses the Chirpstack framework and PyCom devices to implement the proposed proof-of-concept and the codes that were used are provided in Appendix I and II for further exploitation. Moreover, this chapter dives deep into the roles of all of the components discussed in chapters above, as well as the challenges that arose during the development process.

First, the Chirpstack framework is discussed. This framework consists of multiple separately installed components such as: (1) the application server that is responsible for storing the device data and enabling the security techniques of LoRaWAN, (2) the network server, which is responsible for the network communication and the Network layer security, (3) an MQTT broker, which is called Mosquitto and is responsible for exchanging messages between components and transferring the LoRaWAN packets in MQTT message format from component to component, (4) a database of user's choice (Postgres) for storing the above mentioned application and device data. The Chirpstack framework acts as a cloud application / network server, which is accessible through a user interface and various sockets, and where a device sends data and performs various actions. For example, a device can send a temperature measurement every minute and a cloud function or

a cloud integration of the Chirpstack framework can perform a change to another device, e.g., an electric water valve can be opened to release water to plants. This specific example showcases a possible critical industry infrastructure, in which an attacker is able to eavesdrop, capture, alter or not alter, and replay in the air the intercepted data for performing unauthorised actions to connected devices, such as valves, pumps, etc. Towards securing the LoRaWAN network from such MITM attacks, a cloud native application was developed, which operates with the Chirpstack framework remotely by utilising a RestAPI and WebSockets connections over TLS and service account credentials.

To implement the remote operation of the Chirpstack framework, research was initially conducted upon the possibility of sending messages to the MQTT broker, which is responsible for saving and transferring data internally to the Chirpstack framework. This endeavour aimed to forcibly change the AppKey in the Chirpstack framework, but it was unsuccessful due to MQTT restrictions. As a second attempt, an investigation was conducted regarding the GRPC protocol that the Chirpstack framework provides to accomplish some system-level activities, but this also resulted in a dead end because of the complexity of the GRPC. Finally, a technique was discovered to remotely operate the Chirpstack framework by using the RestAPI interface exposed by the Chirpstack framework. After some exploration and testing on the RestAPI, system access was gained to the Chirpstack framework, specifically to the application server, with administrator user credentials and a token. The static AppKey was successfully changed with a newly generated key and the first phase of the proposed implementation was completed. All attempts and tests were carried out using the Postman tool, which is used to conduct HTTPS requests and assisted in exploring the possibilities of the Chirpstack framework's RestAPI.

After discovering how to alter the AppKey from the server, the same action was needed to be accomplished on the device remotely, utilising LoRaWAN packets. The idea was to send a downlink acknowledged packet with a new key inside, from the Chirpstack framework application server, through the secure channel of the downlink, which is already encrypted with NwkSKey and AppSKey from the initial Join Procedure. This way the device is the only one who knows the new AppKey along with the Chirpstack framework application server. Upon key arrival, the key is changed and stored to the memory of the device (specifically, to the NAND flash memory of PyCom devices). Then, the device acknowledges the key change to the Chirpstack framework

application server. Upon acknowledgement, the server changes its AppKey and awaits following messages with the new AppKey.

Finally, as the AppKey is altered in the LoRaWAN device and the Chirpstack framework application server, the codes that were developed were subjected to optimization techniques to ensure optimal performance when used in conjunction with one another. The device code can be found in APPENDIX I and the key rolling module code can be found in Appendix II for further exploitation and improvement.

The sequence / flow of actions that the above implementation is described as follows:

1. The main module waits for a message from a specific extreme edge device based on the DevEUI.
2. Upon message arrival the main module changes the AppKey, sends it back to the extreme edge device, and waits for an acknowledgement (ACK Message).
3. After an ACK message from the extreme edge device is received, the main module changes the AppKey on the Chirpstack application server and waits for a new device message.
4. When the extreme edge device wants to communicate with the backhaul again (usually after some seconds or minutes), for sending the new sensor measurement, it is required to re-join the LoRaWAN network because the application server invalidated the previous AppKey and awaits messages with the new AppKey. The device sends a new Join Request message to the backhaul with the new AppKey that is stored in its local NAND flash memory.
5. The server accepts the new Join Request message from the device with the new AppKey and waits for device data in the next time slot. The device then sends data through the secure tunnel opened by the network and application server and then awaits again for the new AppKey.
6. When the server receives the new UP message from the device, it stores device data (e.g., Temperature) and the main module restarts the same sequence for changing keys.

The sequence and flow diagrams presented in Figure 13 and Figure 14, respectively, provide a more detailed picture of the overall design and the proposed system's implementation. It also aids in understanding the primary players, functions, and components that comprise this implementation.

Sequence Diagram

The sequence diagram in Figure 13 shows the enhanced security level that results from utilizing the key rolling module in LoRaWAN networks. The LoRaWAN network components used in this diagram are the extreme edge devices (on the left), the edge device (gateway), the backhaul (application and network servers), and the key rolling module implemented by this thesis. The communication (first iteration) starts with a Join Request which is prone to attack and is transmitted unencrypted through the air. The Join Accept then is transmitted to the extreme edge device through a sequence of authentication actions taken from the network server. Meanwhile, the key rolling module is waiting to listen for new messages from the device. Then, the device sends a data message (Up message), and the key rolling module listens to that message and generates a new AppKey for the device. The message data are stored to the backhaul and the new AppKey is sent to the extreme edge device with an acknowledged Downlink message (Down). The device stores the received AppKey for usage in the next iteration of communication and then responds to the backhaul with an ACK message. Upon ACK arrival, the key rolling module notifies the application server to change the AppKey and the application server waits for the next communication message with the new AppKey. The loop continues infinitely and every device communication is secured by a dynamic generated AppKey. Last but not least, when the AppKey is changed, the extreme edge device is required to re-join the LoRaWAN network; this way the key rolling module enhances the security of communications between extreme edge devices and the backhaul.

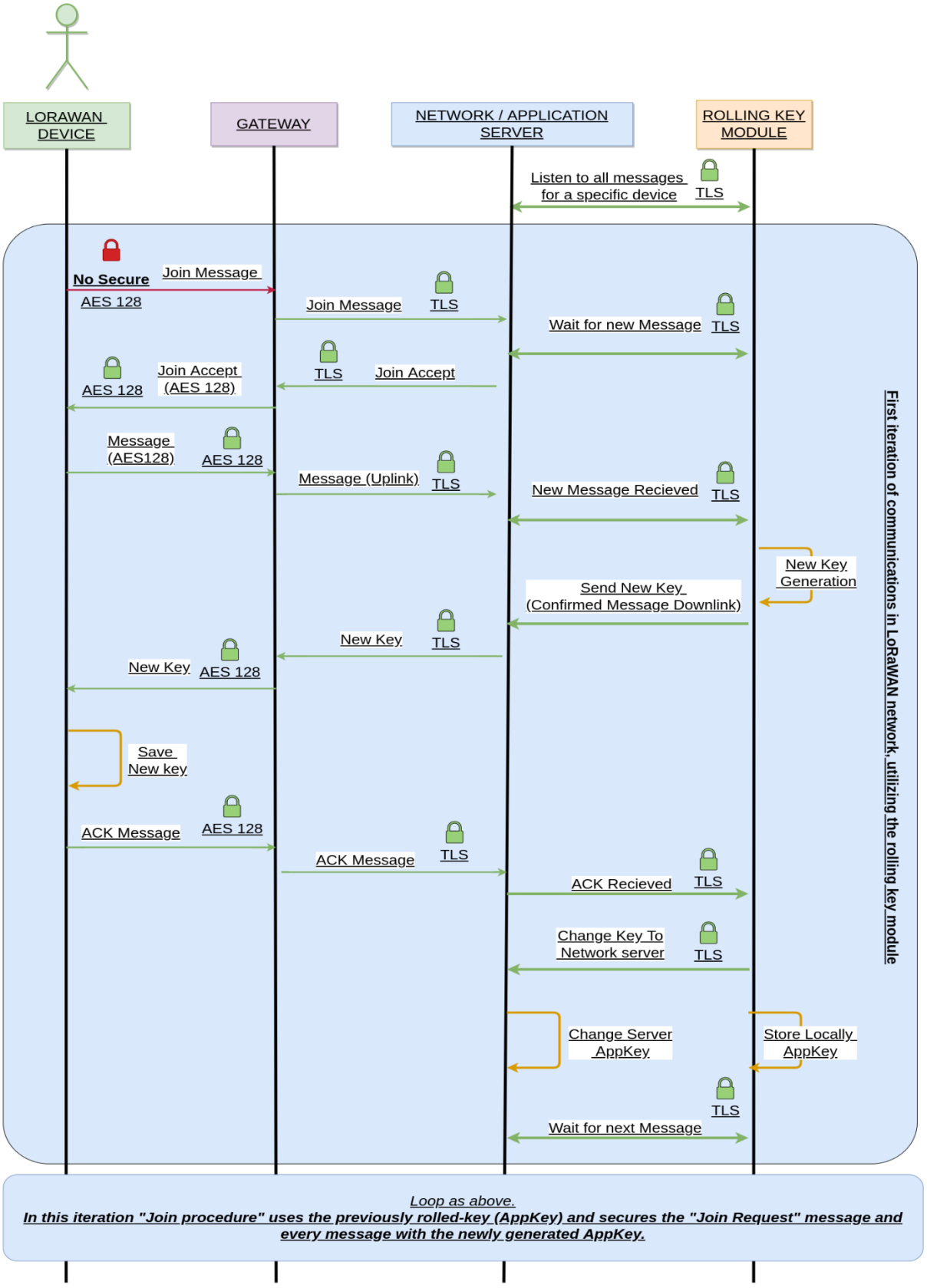


Figure 13 Sequence Diagram

Flow Chart – Diagram

The flow diagram in Figure 14 starts with activation of key rolling module (Start) on top. The containerized microservice of the key rolling module is activated for a specific device by passing environment variables, during the boot process of the container. After the boot process, the key rolling module listens for messages at the backhaul server, through a WebSocket, using the Chirpstack framework's Rest API. Upon message arrival to the backhaul, the key rolling module checks the type of the message. If the type matches the Up-message type, which refers to data sent from an extreme device, the key rolling module generates a new AppKey and sends it to the extreme edge device. The new AppKey is sent in an acknowledgement message to ensure it will not be lost. Upon arrival of the AppKey to the extreme edge device, the device decrypts the message with the previous AppKey and flashes the new AppKey value to its NAND memory for later usage. The device acknowledges the AppKey arrival to the backhaul and the key rolling module, which listens to the acknowledgement message, replaces the AppKey at the application server with the newly created one. By repeating the above-described process infinitely, the LoRaWAN is equipped with a secure key rolling module that generates dynamic keys on every data transmission from devices.

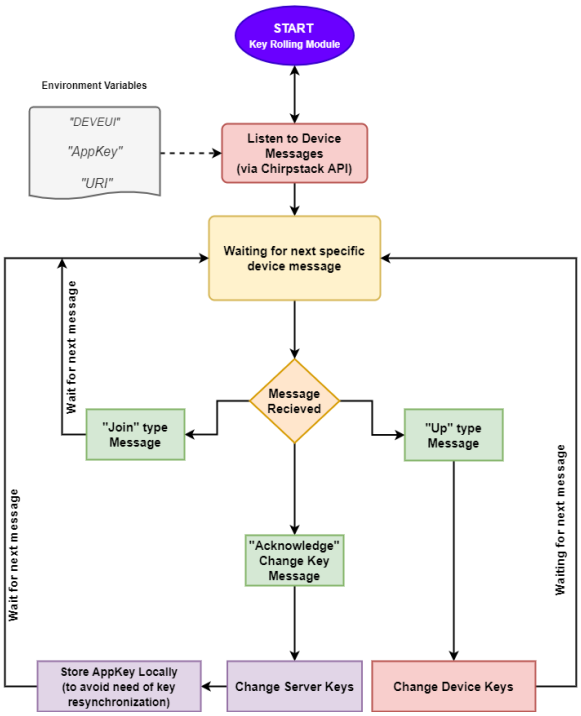


Figure 14 Flow Diagram

Chapter 6 Evaluation

The evaluation of the implemented platform took place in a real-world environment with measurements taken from a private application/network server (from Chirpstack framework) on the backhaul, devices on the extreme edge (PyCom devices) and a privately deployed gateway (Lorix One) on the edge. The evaluation testbed is depicted in Figure 15 and the 3 experiments performed in this thesis system evaluation are named Delay Experiment, Alpha Experiment, and Beta Experiment.

The Delay Experiment measured the key rolling mechanism's performance and the time window introduced by the key change delay, during which an attacker can perform a MITM attack and exploit the vulnerabilities of LoRaWAN networks. The Alpha Experiment measured the impact of an attack on a LoRaWAN network that lacks the integration of the key rolling mechanism. On the other hand, the Beta Experiment measured the impact of an attack on a LoRaWAN network in which the key rolling mechanism is enabled.

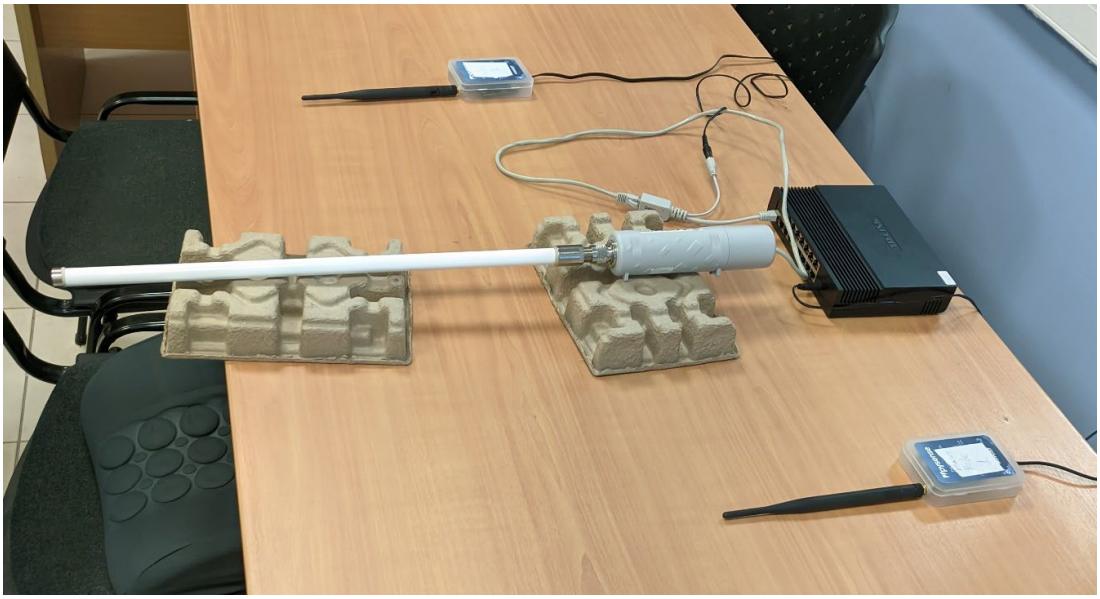


Figure 15 Evaluation testbed

In the Alpha and Beta Experiments, it is assumed that the attacker has already obtained the AppKey from the "victim" device, through either a physical attack (i.e. access to the hardware of the extreme edge device) or by eavesdropping the communication link for aquiring and decrypting the MIC value of the Join Request. Additionally, the attacker is considered to be in close proximity with the extreme edge device in order to perform such attacks. Additionally, for both experiments the "Frame-counter validation", which is an incremental number like DevNonce described in Chapter 1 and is used for ensuring message integrity, was disabled to simplify the attacking method, since the calculation of the frame count (FCNT) value is time consuming process, while it does not violate/cancel the overall concept of the proposed technique.

Delay Experiment

The first experiment measured the key rolling mechanism's performance and the key change delay introduced by the key rolling module code. This delay concerns the total time needed for the following sequential processes: getting an uplink message; generating a new AppKey; distributing the new key to the extreme edge device; changing the AppKey to the Chirpstack application server upon receiving the ACK message from the extreme edge device for AppKey change.
The constant variables of this experiment were:

- extreme edge device packet data: "Temp: 28^{o}C",
- one client every 60 seconds (packet transmission)
- 20 minutes time window for packet capturing (total time of the experiment)

The duration of 20 minutes for the time window was selected because shorter and a longer time window duration did not provide enough measurements for distinct results or provided more than enough measurements, respectively. The client transmission period of 60 seconds is selected because this allows received packets to be distinguished more easily.
The Delay Experiment resulted in an average key rolling delay of about 116 milliseconds for when no attack was performed and in a delay of about 113 milliseconds when an attack was initiated to the network, as depicted in Figure 16.

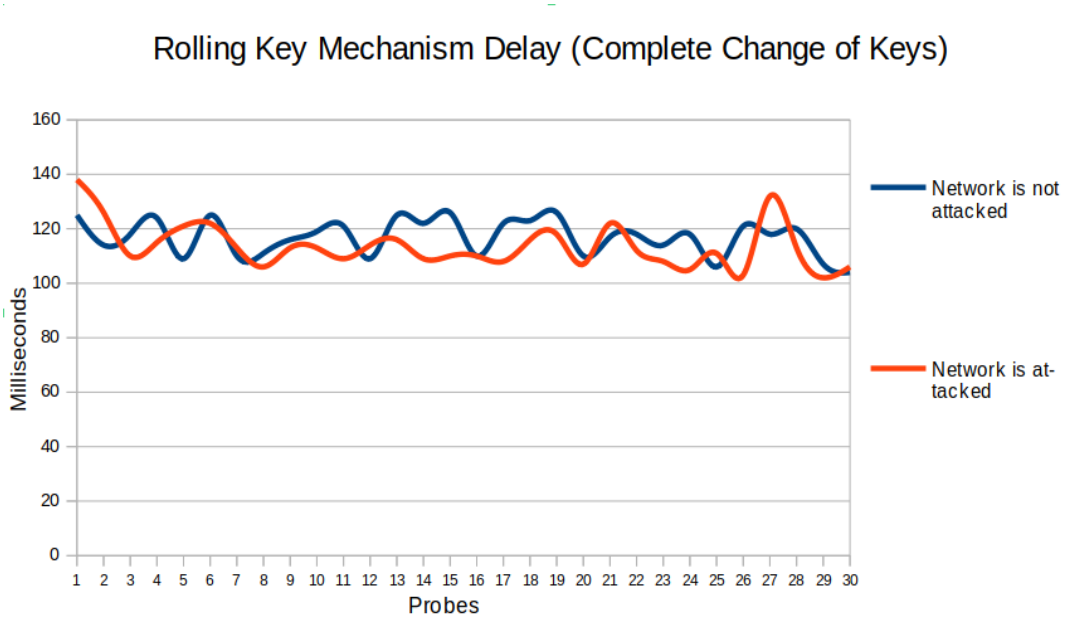


Figure 16 Delay Experiment Results

The results above show that **no effect** is caused to the key rolling mechanism during an attempt of attacking the LoRaWAN network, as there is no additional delay introduced to the procedure. Additionally, this measurement showcases the time window in milliseconds during which the LoRaWAN network is prone to replay and impersonation attacks (MITM attacks) before the AppKey is changed. An attacker is not able to perform such a time-consuming attack in a time window of 116 or 113 milliseconds, because the attacker's device shall have great processing power to intercept a packet, decrypt it, and re-transmit it through the air to the gateway. Moreover, the proposed mechanism adds an extra layer of security in case the application server has disabled the "Frame-counter validation" mechanism. Consequently, this experiment proves that this thesis' proposed mechanism enhances the security of the LoRaWAN networks concerning replay MITM attacks.

Alpha and Beta Experiment

The Alpha and Beta Experiments were performed in a real-world environment, with two main phases for each experiment. These two phases were: (1) Without Attacker and (2) With Attacker. The Alpha Experiment makes use of a LoRaWAN network of version 1.0.2 without the use of the key rolling mechanism developed in this thesis. On the other hand, in the Beta Experiment the proposed key rolling mechanism was integrated in the LoRaWAN network that was utilised.

Both Alpha and Beta Experiments were performed with common constant variables:

- extreme edge device packet data: “Temp: 32ºC”,
- attacker device packet data: “Temp: 50ºC”,
- one client every 60 seconds (packet transmission)
- 20 minutes time window for packet capturing (total time of the experiment)

Moreover, in the With Attacker phase of each experiment an attacker executes a replay attack by flooding the LoRaWAN network with packets every 4 seconds. The client transmission period of 60 seconds is selected because this allows received packets to be distinguished more easily.

The Alpha Experiment resulted as expected. The **Average Packets per minute (ppm)** measurement, which is showcased in Figure 17, was about 13,42 ppm in the With Attacker phase when a replay attack was performed, in comparison to the Without Attacker phase where a value of about 1.05ppm was measured.

In the case of the Beta Experiment, as shown in Figure 17, results show that the attacker was prevented from sending data to the Chirpstack application server. That is because, even if the attacker intercepted the packet and executed the attack, the AppKey was changed on the Chirpstack Application Server by the time the attacker’s packets were received, therefore, these packets were dropped upon arrival. The average packets per minute captured in this network were 4.19 ppm for the Without Attacker phase and 4.09 ppm for the With Attacker phase.

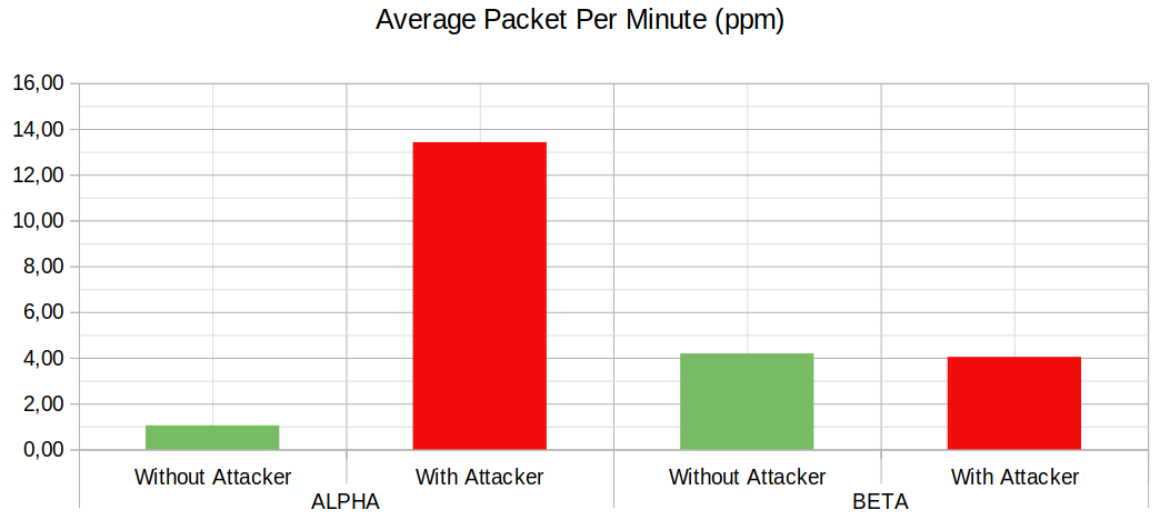


Figure 17 Average packet per minute (ppm)

Moreover, in the Beta Experiment it can be observed that the security solution, proposed in this thesis, prevented the flooding of the storage server with duplicate or manipulated data, because the Chirpstack application server discards the packets with invalid MIC value. The MIC value is invalid because the utilised AppKey is deprecated, and the application server does not expect the previous MIC value. In Figure 18 the number of "Error" packets (packets that were rejected) is depicted. These packets are captured from the application server with the key rolling mechanism as well as from the application server without the key rolling mechanism. The rejected packets of the Alpha Experiment were 0 out of a total number of 282 packets transported in the network. That is, all packets were saved in the application server. Regarding the Beta Experiment, 172 packets were rejected out of a total number of 257 packets transported in the network. This indicates that the key rolling mechanism performed as expected and the attackers were prevented from injecting malicious data or flooding the application server storage.

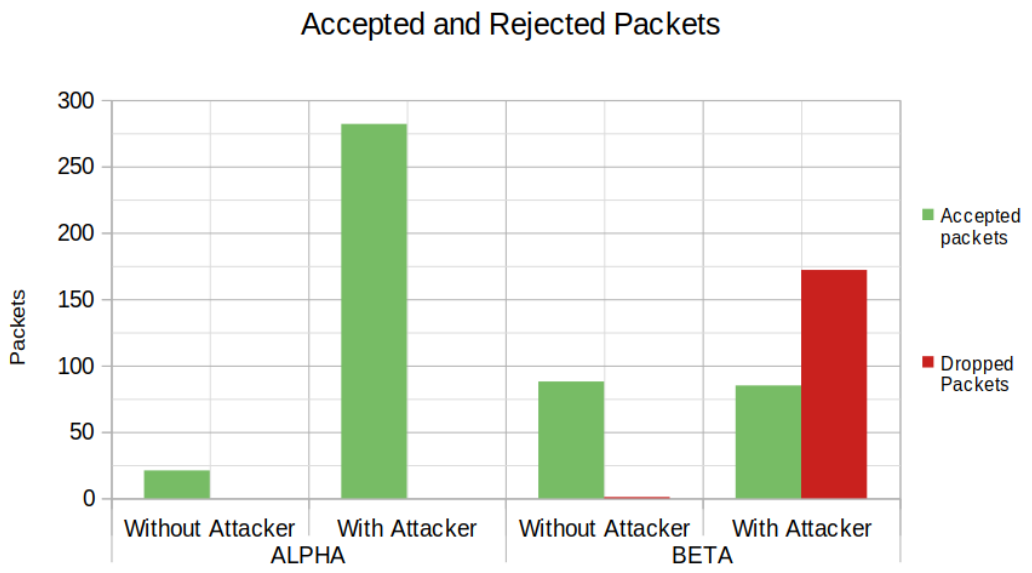


Figure 18 Accepted and Rejected Packets

Based on the results of the experiments that were demonstrated above, LoRaWAN networks using the proposed key rolling mechanism developed in this thesis are reliable, secure, and impenetrable, assuming that the great computing power that is needed for decrypting and reverse engineering the MIC value of a Join Request message cannot be found in COTS devices and Software Defined Radios (SDR).

Chapter 7 Conclusion

This thesis proposed a system that was implemented as a microservice for improving LoRaWAN networks' security by using a key rolling technique similar to the one utilised by the automotive industry for securing car central locking systems. This thesis utilized a LoRaWAN network and PyCom devices for implementing the proposed security mechanism to LoRaWAN networks, for enhancing the security level of the Join Procedure followed by LoRaWAN extreme edge devices operating in OTAA mode. The key rolling mechanism proved to mitigate all open issues related to the Join Procedure, such as the transmission of an unencrypted Join Request message over the air and the use of a static AppKey for device authentication in the backhaul, in versions 1.0.2 and 1.1. By using the key rolling technique, the static AppKey is replaced with a dynamic key that varies with each message exchange session. The short time period required for rolling the new AppKey does not allow attackers to eavesdrop, decrypt, and re-transmit messages with altered content, or perform a replay attack that can result in legitimate user lock-out. Moreover, the evaluation of this thesis in a real-world environment depicted promising results and prevented attackers from jamming the LoRaWAN backhaul. The proposed security mechanism needs further exploration, which is suggested to the LoRa Alliance leading the improvement and standardization of LoRaWAN networks. In the future, a blockchain addition could be implemented for two-factor client authentication that would add an additional layer of security regarding extreme edge device validation. Furthermore, a real time localization, such as RSSI, using various information from multiple gateways could be added for further validation of the extreme edge devices as well as for data integrity assurance.

References

[1] Shanmuga Sundaram, J. P., Du, W., & Zhao, Z. (2020). A Survey on LoRa Networking: Research Problems, Current Solutions, and Open Issues. IEEE Communications Surveys & Tutorials, 22(1), 371–388. https://doi.org/10.1109/COMST.2019.2949598

[2] Tomasin, S., Zulian, S., & Vangelista, L. (2017, May 3). Security Analysis of LoRaWAN Join Procedure for Internet of Things Networks. *2017 IEEE Wireless Communications and Networking Conference Workshops, WCNCW 2017*. https://doi.org/10.1109/WCNCW.2017.7919091

[3] Aras, E., Ramachandran, G. S., Lawrence, P., & Hughes, D. (2017). Exploring the Security Vulnerabilities of LoRa. 2017 3rd IEEE International Conference on Cybernetics (CYBCONF), 1–6. https://doi.org/10.1109/CYBConf.2017.7985777

[4] Kim, J., & Song, J. (2017). A Simple and Efficient Replay Attack Prevention Scheme for LoRaWAN. Proceedings of the 2017 the 7th International Conference on Communication and Network Security - ICCNS 2017, 32–36. https://doi.org/10.1145/3163058.3163064

[5] Isa, H., Bahari, I., Sufian, H., & Z'aba, M. R. (2011). AES: Current security and efficiency analysis of its alternatives. 2011 7th International Conference on Information Assurance and Security (IAS), 267–274. https://doi.org/10.1109/ISIAS.2011.6122831

[6] Biryukov, A. (2005). The Boomerang Attack on 5 and 6-Round Reduced AES. In H. Dobbertin, V. Rijmen, & A. Sowa (Eds.), Advanced Encryption Standard -- AES (pp. 11–15). Springer Berlin Heidelberg.

[7] Butun, I., Pereira, N., & Gidlund, M. (2018). Security risk analysis of LoRaWAN and future directions. Future Internet, 11(1). https://doi.org/10.3390/FI11010003

[8] SeungJae Na, DongYeop Hwang, WoonSeob Shin, & Ki-Hyung Kim. (2017). Scenario and countermeasure for replay attack using join request messages in LoRaWAN. 2017 International Conference on Information Networking (ICOIN), 718–720. https://doi.org/10.1109/ICOIN.2017.7899580

[9] Sung, W. J., Ahn, H. G., Kim, J. B., & Choi, S. G. (2018). Protecting end-device from replay attack on LoRaWAN. International Conference on Advanced Communication Technology, ICACT, 2018-Febru, 167–171. https://doi.org/10.23919/ICACT.2018.8323684

[10] Xia, Z., Zhou, H., Gu, K., Yin, B., Zeng, Y., & Xu, M. (2018). Secure Session Key Management Scheme for Meter-Reading System Based on LoRa Technology. IEEE Access, 6, 75015–75024. https://doi.org/10.1109/ACCESS.2018.2883657

[11] Lin, J., Shen, Z., Miao, C., & Liu, S. (2017). Using blockchain to build trusted LoRaWAN sharing server. International Journal of Crowd Science, 1(3), 270–280. https://doi.org/10.1108/IJCS-08-2017-0010

[12] Danish, S. M., Lestas, M., Asif, W., Qureshi, H. K., & Rajarajan, M. (2019). A lightweight blockchain based two factor authentication mechanism for LoRaWAN join procedure. 2019 IEEE International Conference on Communications Workshops, ICC Workshops 2019 - Proceedings. https://doi.org/10.1109/ICCW.2019.8756673

[13] E. Markakis, Y. Nikoloudakis, E. Pallis and M. Manso, "Security Assessment as a Service Cross-Layered System for the Adoption of Digital, Personalised and Trusted Healthcare," 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), 2019, pp. 91-94, doi: 10.1109/WF-IoT.2019.8767249.

[14] Y. Nikoloudakis, E. Markakis, G. Mastorakis, E. Pallis and C. Skianis, "An NF V-powered emergency system for smart enhanced living environments," 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2017, pp. 258-263, doi: 10.1109/NFV-SDN.2017.8169872.

[15] K. Karampidis, S. Panagiotakis, M. Vasilakis, E. K. Markakis and G. Papadourakis, "Industrial CyberSecurity 4.0: Preparing the Operational Technicians for Industry 4.0," 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2019, pp. 1-6, doi: 10.1109/CAMAD.2019.8858454.

[16] N. Astyrakakis, Y. Nikoloudakis, I. Kefaloukos, C. Skianis, E. Pallis and E. K. Markakis, "Cloud-Native Application Validation & Stress Testing through a Framework for Auto-Cluster Deployment," 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2019, pp. 1-5, doi: 10.1109/CAMAD.2019.8858164.

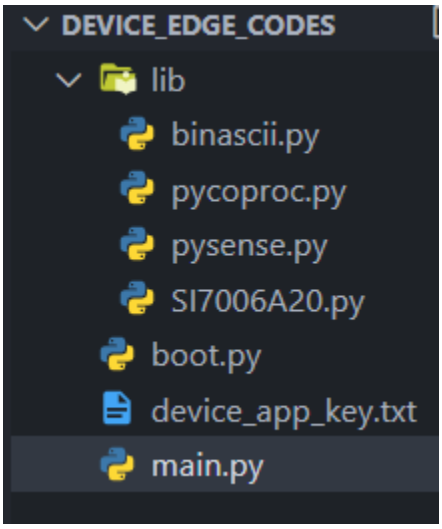
[17] Nikoloudakis, Yannis, Ioannis Kefaloukos, Stylianos Klados, Spyros Panagiotakis, Evangelos Pallis, Charalabos Skianis, and Evangelos K. Markakis. 2021. "Towards a Machine Learning Based Situational Awareness Framework for Cybersecurity: An SDN Implementation" Sensors 21, no. 14: 4939. https://doi.org/10.3390/s21144939

[18] D. Papatsaroucha, Y. Nikoloudakis, I. Kefaloukos, E. Pallis, and E. K. Markakis, “A Survey on Human and Personality Vulnerability Assessment in Cyber-security: Challenges, Approaches, and Open Issues.” arXiv, 2021. doi: 10.48550/ARXIV.2106.09986.

Appendixes

APPENDIX I: End device code

- ***Device folder structure:***



- ***Device Code (Python) (main.py):***
    - Flash devices with PyMakr tool.

```
from network import LoRa
import socket
import ubinascii
import binascii
import PyCom
from PySense import PySense
import time
from SI7006A20 import SI7006A20

py = Pysense()

si = SI7006A20(py) #temp / etc sensor lib

time.sleep(2) # for_reflashing

pycom.heartbeat(False)
```

```
pycom.rgbled(0xff0000)


# It's to parameters the LoraWAN with the frequency of our region
lora = LoRa(mode=LoRa.LORAWAN,
            region=LoRa.EU868,
            power_mode=LoRa.ALWAYS_ON,
            device_class=LoRa.CLASS_A)


dev_eui = binascii.unhexlify('4450b08e04d671b1')
app_key = binascii.unhexlify('00000000000000000000000000000000')
app_eui = binascii.unhexlify('0000000000000000')

# set the 3 default channels to the same frequency (must be before
sending the OTAA join request)
lora.add_channel(index=0, frequency=868100000, dr_min=5, dr_max=5)
lora.add_channel(index=1, frequency=868300000, dr_min=5, dr_max=5)
lora.add_channel(index=2, frequency=868500000, dr_min=5, dr_max=5)

pycom.rgbled(0xff0000)

with open('device_app_key.txt', 'r') as datafile:
   pycom.rgbled(0xff00ff)
   new_key = datafile.read()
   new_key = new_key.replace('\n', '')
   new_key = new_key.replace(' ', '')
   print("Joining with key (Start Up): "+new_key)
   app_key = binascii.unhexlify(new_key)
   datafile.close()


lora.set_battery_level(256) #setting battery to 100%

lora.join(activation=LoRa.OTAA,auth=(dev_eui,                app_eui,
app_key),timeout=0)

if(not lora.has_joined()):
```

```
    new_key = '00000000000000000000000000000000'

changedKey = False

while True:


    if(changedKey == True):
        with open('device_app_key.txt', 'r') as datafile:
            pycom.rgbled(0xff00ff)
            new_key = datafile.read()
            new_key = new_key.replace('\n', '')
            new_key = new_key.replace(' ', '')
            app_key = binascii.unhexlify(new_key)
            datafile.close()

        pycom.rgbled(0xaa5500)
        print("Joining with key (When Key Changed): " + str(new_key))

        lora.join(activation=LoRa.OTAA,auth=(dev_eui,        app_eui,
app_key),timeout=0)

        if(lora.has_joined()):
            changedKey = False

    while (not lora.has_joined()):
        time.sleep(2)
        print("Not Joined Yet... - Retrying...")
        # time.sleep(2) # Rety after 2 seconds
        pycom.rgbled(0xaa5500)
        lora.join(activation=LoRa.OTAA,auth=(dev_eui,        app_eui,
app_key),timeout=0)

    if(lora.has_joined() == False):
        time.sleep(2)
        pycom.rgbled(0xff0000)
        print("Device Not Joined... - Retrying...!")
```

```
    elif(lora.has_joined() == True):
        # create a LoRa socket
        s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

        # set the LoRaWAN data rate
        s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

        s.settimeout(25)  # configure a timeout value of 25 seconds
        pycom.rgbled(0x02df00)
        changedKey = False
        print("Device Joined...!")
        temperature = str(si.temperature())
        print("Sending Temperature Measurement...")
        s.send(b'Temp: ' + temperature)
        print('Send Temp: ' + temperature)
        try:
            data = b''
            data = s.recv(1024)
            if(data != b''):
                print("DATA")
                print(data)
                data = ubinascii.b2a_base64(data)
                data = data.decode('utf-8')
                app_key = data.replace('\n', '')
                print_app_key = app_key
                app_key   =   ubinascii.unhexlify(app_key.encode('utf-
8'))
                print("\nNew Data (Received): " + str(data))
                print("\nNew     App_key     (Received):     "     +
str(print_app_key))
                with open('device_app_key.txt', 'rw') as datafile:
                    new_key = datafile.write(data)
                    pycom.rgbled(0xff00ff)
                    changedKey = True
                datafile.close()

                print("Joining   with   key   (After   Key   Save   in
Filesystem): " + data)
```

```
                lora.join(activation=LoRa.OTAA,auth=(dev_eui,
app_eui, app_key),timeout=0)

        except socket.timeout:
            print('\nNo packet received')

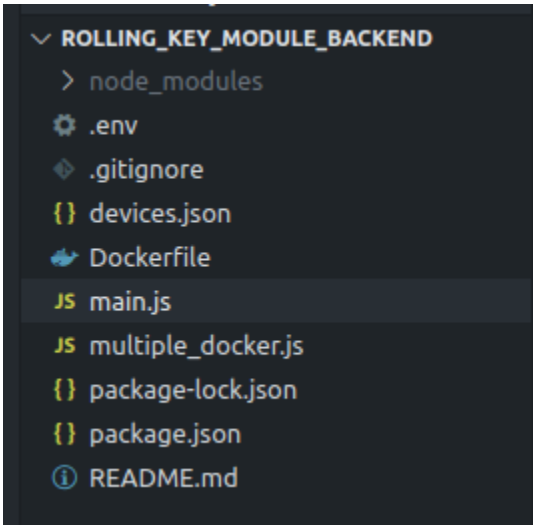
        s.close()
        print("Sleeping 30 seconds")
        pycom.rgbled(0xffffff)
        time.sleep(30)
```

- **device_app_key.txt** file:

```
00000000000000000000000000000000
```

APPENDIX II: Rolling key module code

- ***Device folder structure:***



- ***Main Rolling Key module code (main.js):***

```
const axios = require('axios');
const https = require('https');
const WebSocket = require('ws');
require('dotenv').config();
var crypto = require('crypto');
```

```
const fs = require('fs');

process.env.NODE_TLS_REJECT_UNAUTHORIZED = '0';

var     email_chirpstack     =     process.env.CHIRPSTACK_EMAIL     ||
"test@example.com";
var pw_chirpstack = process.env.CHIRPSTACK_PASSWORD || "password";

var start_change_keys_time;
var change_device_keys_start;
var change_device_keys_end;
var end_change_keys_time;

var in_progress_flag = false;

const agent = new https.Agent({
   rejectUnauthorized: false
});

const dev_eui_regex = new RegExp('^[0-9a-fA-F]{16}$')
const sync_key_regex = new RegExp('^[0-9a-fA-F]{32}$')

var previous_was_join = 0;
var previous_up = 0;


var chirpstack_url = process.argv[2] || process.env.CHIRPSTACK_URL;
var dev_eui = process.argv[3] || process.env.LORA_DEVEUI;

var latest_key = process.argv[4] || process.env.LORA_SYNC_KEY;

var sync_key = process.argv[4] || process.env.LORA_SYNC_KEY;

if (!(chirpstack_url != undefined && dev_eui_regex.test(dev_eui) &&
sync_key_regex.test(latest_key))) {
   console.log("Invalid Arguments");
   console.log("node main.js   < CHIRPSTACK_URL > < LORA_DEVEUI > <
LORA_SYNC_KEY >");
```

```
    console.log("Connecting URL: " + chirpstack_url)
    console.log("Connecting Device: " + dev_eui)
    console.log("Sync Key: " + sync_key)

    process.exit(0);
}

console.log("Connecting IP: " + chirpstack_url)
console.log("Connecting Device: " + dev_eui)
console.log("Sync Key: " + sync_key)

if (chirpstack_url && dev_eui && sync_key) {
    login(chirpstack_url)
        .then((resp) => {
            console.log("Got Chirpstack Bearer Token: " + resp)
            return resp;
        })
        .then(async function (bearer) {
            sync_key_with_server(chirpstack_url,   bearer,   dev_eui,
sync_key);
            listen_events_of_device(chirpstack_url, bearer, dev_eui);
// Opening Websocket to listen device messages
        })
        .catch((error) => {
            // console.log(error);
            console.log("Login Error please check your credentials.");
            process.exit(0);
        })
}
else {
    console.log("CHECK ENVIRONMENT VARIABLES SOMETHING IS WRONG!")
    process.exit(0);
}


async function login(chirpstack_url) {
    const body = {
        email: email_chirpstack,
```

```
        password: pw_chirpstack
    };

    const options = {
        agent: agent,
        headers: {
            'Content-Type': 'application/json'
        }
    }

    var response = await axios.post(chirpstack_url +
"/api/internal/login", body, options);

    return response.data.jwt
}

async function get_keys(chirpstack_url, bearer, dev_eui) {
    var response = await axios.get(chirpstack_url + "/api/devices/" +
dev_eui + "/keys", {
        headers: {
            "Authorization": "Bearer " + bearer
        }
    })
    return response.data.deviceKeys;
}

async function listen_events_of_device(chirpstack_url, bearer,
dev_eui) {


    var ws = new WebSocket("ws://" + chirpstack_url.replace("http://",
"").replace("https://", "") + "/api/devices/" + dev_eui + "/events",
        "Bearer, " + bearer
    );

    ws.on('open', function open() {
        console.log('Listening Messages From Device: ' + dev_eui);
    });
```

```
   ws.on('close', function close() {
       console.log('Disconnected Listening Messages of Device: ' +
dev_eui);
   });

   ws.on('message', async function incoming(data) {
       var                          devEUI                          =
JSON.parse(JSON.parse(data).result.payloadJSON).devEUI;
       const buffer = Buffer.from(devEUI, 'base64');
       const dev_eui = buffer.toString('hex');
       var data = JSON.parse(data).result;

       if (data.type.toLowerCase() == 'up') {
           previous_was_join = 0;
           previous_up += 1;
       }

       console.log("\n"    +    data.type.charAt(0).toUpperCase()    +
data.type.slice(1) + " message from: ", dev_eui)

       if (data.type.toLowerCase() == 'join') {
           previous_was_join += 1;
           previous_up = 0;
       }

       if (data.type.toLowerCase() == 'up' && previous_was_join < 1
&& previous_up <= 1) {
           latest_key   =   await   change_device_keys(chirpstack_url,
bearer, dev_eui)
           console.log("Device Changed Response:", latest_key)
       }

       if (data.type.toLowerCase() == 'txack') {
           change_device_keys_end = Date.now();
           console.log("change_device_keys_end:                    ",
change_device_keys_end)
           in_progress_flag = false;
```

```
            var response = await change_server_keys(chirpstack_url,
bearer, dev_eui, latest_key)
            console.log("Server Changed Keys Response: ", response)
            previous_was_join = 0;
        }


    });

}


async function change_server_keys(chirpstack_url, bearer, dev_eui,
latest_key) {
    const options = {
        agent: agent,
        headers: {
            'Content-Type': 'application/json',
            "Authorization": "Bearer " + bearer
        }
    };
    server_change_body = {
        "deviceKeys": {
            "devEUI": dev_eui,
            "nwkKey": latest_key,
            "appKey": "00000000000000000000000000000000",
            "genAppKey": "00000000000000000000000000000000"
        }
    };


    var response = await axios.put(chirpstack_url + "/api/devices/" +
dev_eui + "/keys", server_change_body, options);
    if (response.status == 200) {
        end_change_keys_time = Date.now();
        console.log("end_change_keys_time: ", end_change_keys_time)
        console.log("KEY: ", latest_key, " -- ", response.status)
        return "Keys changed on server"
```

```
    }
}

async function change_device_keys(chirpstack_url, bearer, dev_eui) {

    start_change_keys_time = Date.now();
    console.log("start_change_keys_time: ", start_change_keys_time)

    if (in_progress_flag == false) {
        in_progress_flag = true;
        var keys = await get_keys(chirpstack_url, bearer, dev_eui);
        var new_key = crypto.randomBytes(16).toString('hex');
        keys.nwkKey = new_key;
        console.log("Changing Key...");
        console.log("Key: ", keys.nwkKey + '\n');

        const options = {
            agent: agent,
            headers: {
                'Content-Type': 'application/json',
                "Authorization": "Bearer " + bearer
            }
        };
        const device_change_body = {
            "deviceQueueItem": {
                "confirmed": true,
                "fPort": 1,
                "data": keys.nwkKey,
                "devEUI": dev_eui
            }
        }


        console.log("New Keys: " + device_change_body.deviceQueueItem.data)
            var response = await axios.post(chirpstack_url + "/api/devices/" + dev_eui + "/queue", device_change_body, options);
            if (response.status == 200) {
```

```
                change_device_keys_start = Date.now();
                console.log("change_device_keys_start:              ",
change_device_keys_start)
                console.log("Started   Key   Change   for   device",
response.data);
                return keys.nwkKey;
        }
    }
    return;
}



async function sync_key_with_server(chirpstack_url, bearer, dev_eui,
sync_key) {
    const options = {
        agent: agent,
        headers: {
            'Content-Type': 'application/json',
            "Authorization": "Bearer " + bearer
        }
    };

    const server_change_body = {
        "deviceKeys": {
            "devEUI": dev_eui,
            "nwkKey": sync_key,
            "appKey": "00000000000000000000000000000000",
            "genAppKey": "00000000000000000000000000000000"
        }
    };
    console.log("...SYNCING KEYS...")
    var response = await axios.put(chirpstack_url + "/api/devices/" +
dev_eui + "/keys", server_change_body, options);
    if (response.data == {}) {
        console.log('Log: Keys changed on server')
        return "Keys changed on server"
    }
```

```
}
```

- ***Device Data Json Analyzer Extracted from Chirpstack (Evaluation):***

```
import json
import sys


def main(input_file: str):
    total_errors: int = 0
    total_packets: int = 0
    total_accepted_packets: int = 0
    avg_packets_per_minute: float = 0.0
    avg_packets_per_second: float = 0.0

    minute: int = 0
    total_minutes: int = 0
    second: int = 0
    total_seconds: int = 0

    # Open File
    with open(input_file) as f:
        data = json.load(f)
        for i, entry in enumerate(data):
            # Count Packets
            total_packets = total_packets + 1
            if entry["type"] == "error":
                # Count Errors
                total_errors = total_errors + 1
            else:
                total_accepted_packets = total_accepted_packets + 1
                only_time = entry['payload']['publishedAt'].split("T")
                split_time = only_time[1].split(":")

                # print(f"TIME: {only_time}")
                # print(f"Split_Time: {split_time}")
                # print(f"Seconds: {split_time[2].split('.')[0]}")

                if minute != split_time[1]:
                    minute = split_time[1]
                    total_minutes = total_minutes + 1
```

```
                else:
                    pass

                if second != split_time[2].split(".")[0]:
                    second = split_time[2].split(".")[0]
                    total_seconds = (total_seconds + 1 )  # multiplied
by 4 seconds that the attackre is sending data to normalize.
                else:
                    pass

    avg_packets_per_minute = float(total_accepted_packets /
float(total_minutes))
    avg_packets_per_second = float(total_accepted_packets /
float(total_seconds))

    print(f"[x] Total Minutes: {total_minutes}")
    print(f"[x] Avg. Number of packet per seconds:
{avg_packets_per_second}")
    print(f"[x] Avg. Number of packet per minute:
{avg_packets_per_minute}")
    print(f"[x] Total Accepted Packets: {total_accepted_packets}")
    print(f"[x] Total Dropped Packets: {total_errors}")


if __name__ == '__main__':
    # python3 ./main.py ./no_rolling_attacked.json
    print(f"[x] Showing results for file: {sys.argv[1]}")
    main(sys.argv[1])
```

- ***Rolling key "Delay" Experiment data analyser (Evaluation):***

```
var fs = require('fs');
var moment = require('moment');
const { parse } = require('path');

var file = process.argv[2];

// TOTAL KEY ROLLING DELAY
fs.readFile(file, (err, data) => {

    var total_rolls = 0;
    var json_data = JSON.parse(data);
```

```
    var total_roll_delay = 0;
    json_data.forEach((datum, i) => {
        if (i < 50) {
            if (datum.ROLLING_START_TIME && datum.ROLLING_END_TIME) {
                total_rolls += 1;
                var delay = datum.ROLLING_END_TIME -
datum.ROLLING_START_TIME;

                // avg calc
                total_roll_delay = total_roll_delay + delay;
                console.log("Complete Rolling Key Delay: ", delay)
                if(delay > 150)
                {
                    console.log(delay, i)
                }
                if(delay < 100)
                {
                    console.log(delay, i)
                }
            }
}
    })

    console.log("FINAL: ", total_roll_delay/total_rolls)
})
```

APPENDIX III: Instructions for deploying and setup of Chirpstack framework

```
# Chirpstack Instructions

### Run chirpstack with docker-compose

```
docker-compose up -d   (For running in Background the Chirpstack framework)


docker-compose up       (For running in Foreground the Chirpstack framework)
```


Open UI:

```
Go to localhost:8080

```

Defaults Credentials:

```
Username: admin
Password: admin
```


Creating a new Gateway
- Go to `Network Servers tab`:
   - Create new `Network Server`:
       - Name: `Whatever you want`
       - URL: `chirpstack-network-server:8000`
       - Don't edit `Gateway Discovery` or `TLS Certificates`
       - Click `Add NETWORK SERVER`
```

```
- Go to `Service Profiles`:
   - Create new `Service Profile`
   - Network Server: `Select the name of nameserver, that was created
in previous settings`
   - Check `Add gateway meta-data`
   - Check `Enable network geolocation`
   - Click `Create Service Profile Button`

- Go to `Gateway-profiles`:
   - Create new `Gateway Profile`
   - Name:  `Name it as you like`
   - Stats Interval: `30`
   - Enabled Channels: `0, 1, 2, 3, 4,5,6,7,8,9,10,11,12,13`
   - Network Server:  `Select the name of nameserver, that was created
in previous settings`

- Go to `Gateways`:
   - Create new `Gateway`
   - Name: `A gateway Name of your choice`
   - Description `A Description of your choice`
   - Click "Refresh Button" on Gateway ID:  `This will generate a
random ID for your Gateway`
   - Network Server: `Select the name of nameserver, that was created
in previous settings`
   - Service Profile: `Select the service profile, that was created
in previous settings`
   - Check: `Gateway discovery enabled`
   - Gateway Altitude: `120` (add your own altiture - optional)
   - Marker on map: `Move marker to your location on map`
   - Click `Create Gateway`



### Programming Gateway Settings

- Go to file `PROGRAMMING_GATEWAY.md` for more instructions
```

APPENDIX IV: Instructions for flashing / programming gateway

```
# Programming LORIX One Gateway to connect with Chirpstack

# 1st Reset Lorix One

- Press The `Reset` button under the `Lorix One Antenna` for at least 20 seconds and after release...

- Reset may take 3-5 minutes.

---
---
# 2nd Find Lorix One IP


- Go to a browser and type `https://lorix-one-0aa218.local/`

   ##### - - - or - - -


- Find IP with `Nmap` or `arp -a` and then `navigate to IP with a browser`.


---
#### Default Username and Password Are:

- Username: `admin`

- Password: `lorix4u`

---

# 3rd Configuration of Lorix One Antenna for Chirpstack

##### You should see a message on top:
```

```
`The antenna type is not yet configured. Please tell us what you've
connected so the LoRa forwarders can be set up properly.`

---

- Click the button on it... `Configure` and select GAIN corresponding
to your antenna (`4dbi antenna for Lorix One in Pasiphae - The antenna
looks like a Lightsaber from star wars`).

- `Update System` before continuing if a new version is available.
(Go to System from Sidebar and Update...)

- Change `admin` user's `password`...

---

- Go to `Network > Ethernet`

 - Click `Edit Settings` (Pencil Button)
 - Add `Static IP Address `For Gateway
   - (There is a "here" underscored text stating about an untrusted
certificate when applying a new configuration... Press it in case
your IP does not change and configuration rollback...)

---

- Next we go to `LoRa` tab from sidebar....

 - Click `Forwarder`
 - Edit `Forwarder Settings`
 - Select `Chirpstack Gateway Bridge Forwarder`
 - `Apply` Settings

---

- Next go to forwarder `Bridge Configuration`
```

```
 -                              Check                              file
`./LorixConfigs/bridgeConfiguration/bridgeConfiguration.conf`
 - Click `Edit` configuration and apply your configuration to the
Bridge...
   - Edit `gateway_id` to `match` the one generated in `Chirpstack`
`Setup`!
   - Edit `MQTT IP ADDRESS` to match your backhaul.
   - Edit `MQTT USERNAME AND PASSWORD` if changed

---

- Go to `Backend configuration`

 - There you have `2 Configurations` in `TABS`

 - Local Configuration
   - Change `GATEWAY ID` to `match` the `one set` `in chirpstack`
backhaul...
   - Change `FAKE GPS COORDINATES ONLY IF NEEDED`

 - Global Configuration
   - Change `lorawan_public` to `true`
   - Change `server_address` to `match` the chirpstack backhaul
   -  Change `port_up` and `port_down` to `match` the chirpstack
backhaul
   - Change Timing parameters as needed (optional)
   - Change Invalid packets forwarding rules (optional)


## FINISHED!
## NOW YOU CAN START FORWARDER AND WATCH LOGS FOR CONNECTION TO
BACKHAUL!

## YOU CAN GO TO http://chirpstack_ip:8080/#/dashboard and CHECK IF
LORAWAN GATEWAY IS AVAILABLE!
```

APPENDIX V: Instructions for installing devices to Chirpstack framework based on DevEUI

```
# ⚡ Security As A Service Lora - Rolling Key Module (Node - LoRa Node - Device Side) (📲 Instructions for Devices)


## 1st Step:
#### SETUP DEVICE TO CHIRPSTACK BACKHAUL
```
- Go to `Device-profiles` -- You need to do this only once! --
   - Setup a `device profile` for `all LoRaWAN EDGE Devices`
   - Options:
       - Network Server (`Select network server setup in Chirpstack Setup Instructions`)
       - Latest LoRaWAN MAC version (`1.0.2`) (Depends on Edge Device Support)
       - LoRaWAN Regional Parameters revision (`A`)
       - ADR algorithm (`Default Algorithm`)
       - Uplink Interval Seconds (Depends on pycom programmed sleep time) (`Setup as required`) (This is `used to determine` if a `device` is `active` or `inactive`.)
   - Go to `JOIN (Tab)`
       - Click `Device Supports OTAA`
   - Go to `Class Tabs`
       - Click `Device Supports Class B` and `Class C` (For Pycom Devices - may defer in other devices)
   - Apply settings and `create device profile`

- Go to `Applications`
   - Create application
       - Add `name, description and select service profile` you created in Chirpstack SETUP instructions before Gateway configuration.
```
```

```
   - `Click on Application name`, `after creation` and it will `lead you to a menu` with tabs `"Devices, Multicast Groups, Integrations, etc.."`

- Go to `Applications > Click Application's name > Devices Tab`
   - `Create a Device`
       - Add a `name` for Edge Device (`Node-1`)
       - Add a `description` for Edge Device
       - Add your device `MAC Address` in `DEVICE EUI` (e.g `44 50 b0 8e 04 d6 71 b1`) or Generate one (`Not Recommended!`) (Generated shall be `WORLDWIDE unique`!)
   - Click `Create`

- Next go to device keys `Applications > Click Application's name > Devices Tab > Click on Device's name > Keys (OTAA)`
   - Add `application key` (APPKEY) as `00 00 00 00 00 00 00 00`, so it will be `better` for you when `integrating` the `key-rolling` module in later steps.
   - Click `Set Device-Keys`
```

2nd Step:
Programming Device...
```
- Requirements:
 - VSCode Pymakr Extension
 - Device EUI
 - AppKey (SyncKey) (Generated from Chirpstack Application Server)
```

Changes in code and settings:
```
- Open `Device_Edge_Codes` folder in `VSCode`
```

```
- Go to `device_key.txt` and change the key to `00000000000000000000000000000000` as set up in the application before `Without Spaces`!

- Next, Go to `main.py` and set following settings as required:
   - The `dev_eui` should match the MAC Address of `device` and `application server` > device that we created in previous steps`!
   - The `app_key` and `app_eui` shall match the keys setup before in `application server` without spaces  (e.g `00 00 00 00 00 00 00 00``) !
```
```
dev_eui = binascii.unhexlify('70B3D5499C5FC7B5')
app_key = binascii.unhexlify('00000000000000000000000000000000')
app_eui = binascii.unhexlify('0000000000000000')

# set the 3 default channels to the same frequency (must be before sending the OTAA join request)
lora.add_channel(index=0, frequency=868000000, dr_min=5, dr_max=6)
lora.add_channel(index=1, frequency=868000000, dr_min=5, dr_max=6)
lora.add_channel(index=2, frequency=868000000, dr_min=5, dr_max=6)

```
Last:
```
- Leave `boot.py empty`!
```

3rd Step:
BEFORE Flashing Device!
⚠ PREPARE TOOLS AND DEVICE:
```
   - RUN COMMAND (This enabled serial access to user): ` sudo usermod -a -G dialout $USER `
   - Open `VSCODE` and install the `pymakr` extension!
   - DO NOT PRESS DOWNLOAD IN PYMAKR this will destroy your code (it downloads code from device if connected!)!
```

```
   - ⚠ Update device firmware (https://docs.pycom.io/updatefirmware/device/)
      - It's better to update firmware with pycom-fwtool before continuing further!)
   - Drivers can be found here if needed: (https://docs.pycom.io/gettingstarted/software/drivers/) !
```

4th Step:

⚠ Flashing Device:

```
- Connect Device to PC with USB
- Check PyMakr at the bottom of vsCode!
- Press Upload!
```