LoRaRatchet, implementation and investigation

Msc thesis

David Martin Carl
daca@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Supervisor: Alessandro Bruni, *brun@itu.dk*

STADS code: KISPECI1SE

Computer Science
IT University of Copenhagen
Denmark
June, 2022

Abstract

The intent of this thesis was to develop and document the development of the LoRaRatchet protocol as a proof of concept. Throughout this report the reasoning why LoRaWan isn't ideal is explained, why LoRaRatchet could be an interresting alternative and the process of implementing a proof of concept of LoRaRatchet. The process involves explaining design decisions and changes regarding the implementation of LoRaRatchet as the messages were based upon the LoRaWan message, and they werent used. The message was also constructed to omit some fields which werent relevant for our experiments. We will also explain how our 2 libraries are implemented and some of the inner workings of these libraries.

The proof of concept was then used to run on a ESP32 as a constrained device, but due to some issues the LoRa communication was based on a Raspberry Pi 3b+. Our experiments first showed that the LoRaRatchet protocol implemented in the Rust Language is a viable option, and how the battery life is impacted by the diffie-hellman ratchet intervals, and how it is tied together with message loss.

The experiments showed a higher power consumption if there was a message loss than first expected. This seemed to be related to factors that were not directly tied into the ratcheting itself, but rather how LoRaRatchet was specified to function with receive windows and listening for messages. The experiments also showed the impact a lower diffie-hellman ratchet interval would have on the battery usage, due to having to perform a higher amount of ratchets.

From looking at the experiment data, I also tried to find a most optimal diffie-hellman ratchet interval based on a few parameters as discussed, this is such as message loss, and how important security is prioritised compared to battery life. From what is seen this looks to be either between a interval of 64, lower if security is very important this would further impact battery life or 256, if battery life is more important but security is still a wanted property.

This abstract was written after the group split up.

Acronyms

ADR Adaptive data rate
AEAD Authenticated encrpytion
APPEUI Application EU identifier
AS Application server
CBOR Concise Binary Object Representation
COAP Constrained Application Protocol
COSE CBOR Object Signing and Encryption
DH Diffie Hellman
ECDH Elliptic Curve Diffie Hellman
Fcnt Frame counter
DEVEUI Device EU Identifier
DHRP Diffie Hellman Ratchet Procedure
HKDF HMAC key Derivation Function
KDF Key Derivation Function
kid Key identifier
LoRa Long-Range
mic Message integrity code
Mtype Message Type
MK Message key
OSCORE Object Security for Constrained RESTful Enviorments
PRK Pseudo Random Key
JS Join Server
RCK Receiving chain key
RK Root Key
RNG Random Number Generator
SCK Sending chain key
SPI serial peripheral interface
TSR Transmission Succes rate

Contents

1 Introduction

The field of IoT has become more and more prevalent in the past decade. In 2010, 0.8 billion IoT & Low Power Wide Area Network (LPWAN) device connections were registered in a market analysis [27]. In 2020, this number was 11.7 billion connections, and estimated at 30.9 billion in 2025 [27].

The LPWAN subset of IoT architectures, have seen unprecedented growth over the past seven years [27]. Networks such as LoRa (Long Range) are optimized for functioning at long range, consuming low power, and being cheap. This is ideal for use-cases in which cheap devices should be functioning for a long time, while not necessarily being geographically close to other nodes. [33, p.2-4].

LoRaWAN is often used as a security layer on top of of LoRa. LoRaWAN uses hardcoded keys to generate session keys between parties, that lasts the lifetime of the device.[24, p.8].

One concern that stands out though with LoRaWAN, is how symmetric session keys are derived from symmetric long-term root keys, where root keys are used for the lifetime of the device. This compromises properties like forward security, recoverability and key renewal [33]. Since the same key is used for encryption of all payloads, the disclosure of a single key would completely compromise all future and past messages.

In the summer of 2021, a new protocol, LoRaRatchet protocol was proposed by Astrid Ellermann-Aarslev and Laura Kromann-Larsen. in an attempt to mitigate some of these issues. The LoRaRatchet protocol is inspired by the well-known end-to-end encrypted chat protocol, Signal, and adopts some of it's features, such as a Double Ratcheting scheme for key-rotation, ensuring that the same encryption key is never used twice.

The loRaRatchet protocol makes some key changes to the Signal protocol in order to let it comply with regulations and constraints for LoRaWAN [8]. One of these changes is the introduction of a revised Double Ratcheting scheme, where a Diffie Hellman Ratcheting procedure (DHRP), can be set to happen at a configurable interval. The DHRP is used to renew renew session keys, and provides the property of Post Ćompromise Security to them.

Some initial findings suggest that while having this procedure happening often may be desirable, to update the key context often, it may also take a toll on the power consumption of whatever constrained device is running the protocol [8, p.79-80].

This this thesis, we attempt realize the LoRaRatchet protocl, by implementing on a constrained device. Our research question is as follows:

When implementing LoRaRatchet on a constrained device, how does the variable DHRP interval affect the power consumption of the device?

We plan to answer our research question by the following steps:

- Specifying which design decisions need to be taken, to implement a working LoRaRatchet instance.
- Implementing the protocol on a constrained device.
- Perform experiments, measuring the power consumption of the device, based on it's set DHRP interval.

In this report its possible to read about how LoRaRatchet was implemented on real hardware, from the hardware aspect into the software aspect regarding the programming of LoRaRatchet.

"In chapter 2, its possible to read about the necessary background information regarding, how LoRa, LoRaWan and LoRaRatchet functions. The security properties necessary to understand why LoRaRatchet is an alternative protocol and some of the underlying mechanism used for LoRaRatchet.

In chapter 3, its possible to read about the different hardware that were used for this project, and why the given hardware was selected.

In chapter 4, the process and steps of implementing the two sub-protocols needed for LoRaRatchet, edhoc and double ratchet are explained. This are done with code examples along the way.

In chapter 5, the experiments we want to conduct are explained in depth in this section. This is our 3 different test setups, how the measurements where taken and what to look out for regarding these experiments.

In chapter 6, the results from the experiments described with the relevant results we achieved from running these experiments."

The quoted part here is written after the group split up.

Disclaimer, this report and work has been done in cooperation with Ask Sejsbo, there is work that has been done before the group split up. Due to the late nature of the split up, everything will be taken as a collaboration effort, unless noted otherwise. There might be smaller changes and corrections which are not quoted but major parts that are done alone will be noted as such, otherwise its written in collaboration with Ask.

2 Background

To understand why the LoRaRatchet protocol was proposed we first need to understand what LoRa and LoRaWAN is, the current issues regarding LoRaWAN, some general security knowledge. There will then be explained a possible solution to the given issues, and the technologies used for the solution.

2.1 LoRa

LoRa works as the physical communication layer of LoRaWAN (but can function as a standalone), and utilizes radio modulation technology, to create long-range communication links.

The rate of transmission is controlled by the chirp rate, also called the spreading factor, which governs how much the signal should vary in frequency. This spreading factor thus controls the rate of data transmission [30]. LoRa uses a spreading factor which can range from values 7 to 12, a larger spreading factor improves the communication range, but spends more energy. LoRa can also use different bandwidths, allowing more data to be sent, and providing better quality of the signal.

The combination of the spreading factor and the bandwidth, is called the data rate, and governs how fast data can be transmitted. A higher data data rate will result in faster transmissions of data, coming at the cost of a lower spreading factor [30]. LoRa enables low power usage, and depending on conditions and data rate, enables long range communications up to 15 kilometers [24].

As the physical layer, LoRa defines the Radio PHY layer, which carries a field for Radio preamble, a physical header (PHDR), a cyclic redundancy check (CRC), and the physical payload (PHYPayload). It is in the PHYPayload, that messages are defined by the protocol running on LoRa.

2.2 LoRaWAN

LoRaWAN (Long Range Wide Area Network) is a protocol that functions as a security layer on top of LoRa. LoRaWAN defines the PHYPayload in of the LoRa fields, and an infrastructure to go with it. LoRaWAN is optimized for battery life of devices, and can provide up to 10 years of lifetime for them. LoRaWAN defines a network structure where gateways propagate messages between end-devices and a centralized network server (NS), which forwards packers to an associated application server (AS). A trusted entity called the join server (JS) is also used to authenticate the parties [23, p.2-5]. The JS, AS, and ED, and each their own IEEE EUI64 adress. The adresses are called the JoineEUI, APPEUI and DEVEUI, respectively.

An end-device (ED) that wants to authenticate itself to the LoRaWAN infrastructure, must undergo a Join procedure (figure 2.1)(or Over-the-air Activation). The Join procedure is the handshake of LoRaWAN, which authenticates the ED to the AS. Messages are mediated through a network server, and a Join server (JS) is used to carry out the Join procedure, which forwards session keys to the application server. LoRaWAN defines its messages in the PHYPayload of the radio PHY layer, that contain information about the message type, the message itself, and a mic tag [24, p.6]. The JS and the ED share the root keys NwkKey and AppKey. At a high level, the Join procedure is a two message protocol that consists of a Join-Request from the ED, that contains some identifying information, and a message authentication tag (mic), The JS then responds with a Join-Accept, which contains an encrypted payload of some nonces, network settings, and a mic tag. After exchanging these two messages, session keys are calculated through AES encryption of the root keys, and the exchanged nonces. A more detailed description can be read at figure 2.1, and in [24]. Whenever any of the parties send a message, they use a value called a frame counter (FCnt) to correlate messages.

After these two messages are exchanged, session keys are derived from root keys and the exchanged nonces [23]. In LoRaWAN, all messages apart from the initial one is validated with a mic tag, and

encrypted under AES.

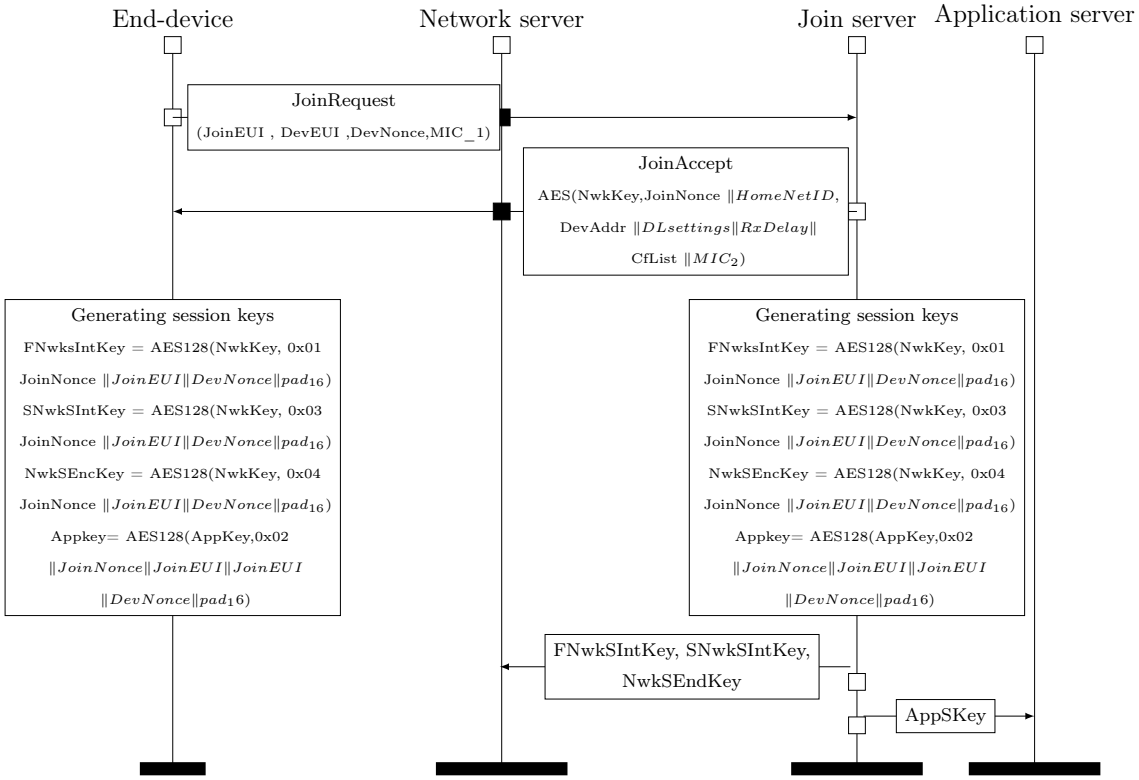

Figure 2.1: LoRaWAN v 1.1 Join procedure

After this procedure, the ED and the AS will in principle be using the same derived session keys for the rest of the ED's lifetime. By convention, messages from the ED to the server is referred to as "Uplink" messages, while the converse case is called "Downlink".

ED's in LoRaWAN usually work by operating in a sleeping state, and waking up in intervals whenever a action should be taken, such as sending a message. After sending a message, the device a receive window where it polls for incoming data. If nothing arrives in the first receive window, it waits for a short while, and opens another one and listen again. These two receive windows are called RX1 and RX2. The duration of the receive windows, usually depend on the conditions the device is deployed in [23, p.14].

2.3 Security considerations of LoRaWAN

The Lora alliance themselves claim to implement end-to-end security through usage of the session key that encrypt payloads, and associated mic values. However, in the relevant literature, several issues with the security of LoRaWAN have been described. In this section, we will describe a few of these.

The first issues arise from the simple fact that root keys are hardcoded onto devices. Any attacker that has access to a device may in some cases be able to read the keys of the hardware. It has also become a popular approach for some providers to provide devices with QR codes that make up the root keys. In these cases, it will be easy for attackers to derive session keys [4].

There are certain areas where LoRaWAN does not adhere to what is usually understood as good practice. An example of this is the usage of ECB mode with AES. Using this mode of encryption is

usually susceptible to pattern leaks, replay attacks, and differential cryptanalysis. LoRaWAN claims to somewhat counter this by including counting nonces with every encryption [32, p.4].

Another attack described in the literature was by Yang et. al. and relies on the fact that the communication between the NS and AS do not share true end-to-end encryption since a mic value does not protect payloads between. Thus no integrity protection is provided to those messages. This opens the door for a bit-flipping attack, where the payloads are modifiable [32, p.41].

The two message join procedure of LoRaWAN has also been shown to provide a weak notion of authentication, as the two parties are not assured that they have generated the same session keys [7].

The underlying issue with LoRaWAN, is its failure to provide forward secrecy as any disclosure of a long-term key will also allow an attacker to derive all past session keys [34].

2.4 Forward security and Post compromise security

The forementioned attacks and exploits are not the subject of this report. But rather they serve as a motivating factor as to why a more secure protocol may be warranted.

A success-full run of any of the mentioned attacks (or others), could lead to disclosure of both future and past sessions. In order to speak more clearly about these properties, or lack thereof, we will define two terms.

Forward security

Forward security (FS) concerns itself with the issue of confidentiality of current messages, if long-term keys are disclosed in the future. An example of this could be an adversary storing ciphertexts sent on the wire, and then later in time obtaining a key, such that the ciphertexts could be decrypted.

Post Compromise security

Post Compromise Security (PCS) goes by many names, it may also be referred to as future secrecy, backwards secrecy or recoverability. In this report, it will primarily be referred to as PCS.

PCS deals with the security of future sessions, in the case that current session keys are disclosed. An example of this property would be an attacker that secretly obtains a session key, and eavesdrops messages sent over the wire, thus compromising all messages in the future.

2.5 A possible solution: LoRaRatchet

In order to mitigate these security issues in LoRaWAN, a new protocol was suggested [8]. The LoraRatchet protocol specifices a communication protocol between ED's and servers that preserves PCS, FS and lives up to stronger notions of authentication [8].

The core idea of the LoRaRatchet protocol is to authorize EDHOC with X3DH. Once a handshake has taken place, a Double Ratcheting procedure is used to ensure FS and PCS for the duration of the communication. Signal is further specified at [17].

In this new protocol, the Join server of LoRaWAN is eliminated, in order to enable a more clear definition of end-to-end encryption, LoRaRatchet abstracts away the responsibilities of the Join Server in the Join Procedure. Thus making the join procedure a interaction between applicaton server and ED, routed through a Network Server (NS) [8, p.66]. LoRaRatchet defines a PHYpayload for LoRa, which complies with EU regulations. This means restricting the PHYPayload to a maximal size of 64 bytes. In order to comply with these regulations, the message sizes required by X3DH was not realizable. For this reason, LoRaRatchet opts for another handshake protocol. In place of X3DH, the Ephemeral Diffie Hellman over COSE (EDHOC) protocol is used. So in LoRaRatchet, the Join procedure consists of the EDHOC handshake, while the double ratcheting mechanism is used for securing messages going

forward.

2.5.1 LoRaRatchet message composition

The LoRaRatchet protocol, is based on LoRaWAN, and has a similar message structure. LoRaRatchet defines a PHYPayload in the same manner as LoRaWAN does. As mentioned, the LoRaRatchet protocol adheres to the smallest allowed PHYPayload size in the European region, which is 64 bytes. The fields of LoRaRatchet's PHYPayload, and their sizes, can be read in table 2.1 [8, p.35].

Field	Mtype/RFU	Nonce	Devaddr	FCtrl	FCnt	FOpts	Fports	FRMPayload	MIC
Bytes	1	13	4	1	2	0..15	1	0...x	8

Table 2.1: PHYpayload message composition

The PHYPayload starts with the **Mtype** field of a byte, which indicates which type of message is coming, optionally, this field also contains three bits that are reserved for later use (**RFU**). Then comes 13 bytes of a **Nonce**, which is typically used as the nonce in the encryption of the application payload. The four-byte **Devaddr** is an ID identifying the ED, and is assigned by the server as the first message is received, it is used as the long-term connection identifier for forward-going communication. The **Fctrl** field holds several bits that specifies information relating to the data rate of the ED. An important field is the Frame counter **Fcnt**, that each party appends to every message, to indicate how many message have been sent, this value plays an important part in the ratcheting part of LoRaRatchet, which is explained in section 2.7.2. When a uplink message is sent, the **Fcnt** is referred to as FcntUp, while in downlink messages it is called Fcntdown.

The **FOpts** field holds information related to network administration, while the **Fport**. Finally, the **FRMPayload**, contains the actual application data that is transmitted, which often will be the encrypted data. At the very end lies the **MIC** field of 8 bytes. The **MIC** field is simply the authenticated part of the ciphertext given by whatever authenticated encryption algorithm is used[8].

In this report's implementation, certain fields are omitted, these fields are described in section 4.1.

The two global IEEE EUI64 adresses **APPEUI** for the AS and the **DEVEUI** for the ED, are stored in the server and device before deployment. They are sent by the ED in the first message, and allows the AS to verify that the message has arrived at the correct adress. [8].

2.6 EDHOC

TLS has become quite dominant for most internet devices as the security layer. However, in the setting of constrained devices do not always have the resources needed to run TLS. For this reason Ephemeral Diffie-Hellman Over COSE (EDHOC) was developed as a lightweight protocol alternative, mostly intended Constrained Application Protocol (COAP) communication that is viable in constrained settings. This makes it a good candidate for embedded devices, saving resources [22]. EDHOC is used as the Join procedure for the LoRaRatchet protocol, and this section will describe the EDHOC protocol, related to it's usage in LoRaRatchet.

EDHOC extends on the family of SIGMA protocols, and is constructed to provide authenticated establishment of keys for IoT environments. It's main use case is to derive a initial context for the Object Security for Constrained RESTful Environments (OSCORE) protocol, a lightweight communication protocol for end-to-end security of COAP messages. EDHOC can however be used with other transports, and for other purposes, such as our LoRa use-case [22, p.4].

CBOR in EDHOC

EDHOC uses CBOR for serialization, which is a binary format, based on JSON, but for embedded environments. Serialized objects in CBOR are referred to as data items. The initial byte of each

data item contain information about the serialized item, where the first 3 bits are called the major type, and contains the type of the contained data. The last 5 bits indicate additional information [2]. CBOR has support for primitive types as byte strings, or integers, and also collections of data items as maps or arrays. As an extension to CBOR, CBOR sequences are also defined, which are simply concatenated data items [1]. EDHOC uses CBOR sequences quite extensively. An example of this in CBOR's diagnostic notation can be seen in listing 2.1.

```
message_1 =
   (
    3, // Method
    0, // Suite
    h'3AA9EB3201B3367B8C8BE38D91E57A2B433E67888C86D2AC006A520842ED5037', // public
    key
    12, // connection identifier
    (1, h'b41a0604620608') // optional ead
   )
```

Listing 2.1: CBOR sequence example (EDHOC message 1)

Utilizing the fact that CBOR items are self-delimiting, CBOR sequences and can thus be used over CBOR arrays, without indicating the entire message length, and thereby reducing message lengths. [1]. For a standard for cryptography, EDHOC uses CBOR Object Signing and Encryption (COSE), which specifies how to provide inputs for cryptographic primitives, or how to represent key credentials [20].

Cryptographic primitives

Implementaions of EDHOC can choose between using digital signatures, or static DH keys for authentication. The signature option forces the messages sent to be upwards of 105 bytes, making it too large for the PHYpayload [22]. The case where both parties use static DH keys for authentication, is referred to as authentication method 3, which LoRaRatchet opts for, for the case of space requirements [8, p.23]. This method requires that each party is initialized with a Elliptic Curve Diffie Hellman (ECDH) keypair, and that the parties have access to the other parties public key. ECDH is a variation of standard Diffie Hellman (DH) key exchange, which allows two parties who both agree on the same elliptic curve to exchange public key, and derive shared secrets, but allowing for smaller key sizes, and faster computation than more conventional DH keys, such as RSA.

EDHOC requires a choice of cryptographic primitives for ECDH keys, authenticated encryption (AEAD), and hashing, used for generation of pseudo random keys (PRK). Authenticated encryption refers to encryption, with built-in integrity checks on the ciphertext, and associated data which can be passed to the algorithm, which is also authenticated.

It is possible to choose between five different cipher suites (0,1,2,3,4),[22, p16]. In LoRaRatchet, the ciphersuite 0 is chosen, which is optimized for smaller message sizes, and can be viewed in table 2.2.

AES-CCM-16-64-128, which is the designated AEAD algorithm refers to using AES encryption in counter mode, which uses a counter to make each block of the encrypted ciphertext different. In conjunction with AES, AES-CCM uses CBC-MAC to generate a message authentication code, providing integrity. AES-CCM-16-64-128 in particular uses a 128-bit key, 64-bit tag, and 13-byte nonce. We will refer to the algorithm as AES-CCM going forward.

EDHOC AEAD algorithm	AES-CCM-16-64-128
EDHOC hash algorithm	SHA-256
EDHOC MAC length in bytes (Static DH)	8
EDHOC Key exchange	X25519
Application AEAD algorithm	AES-CCM-16-64-128
Application hash algorithm	SHA-256

Table 2.2: EDHOC cipher suite 0[24, p.17]

For generation of PRK's, a key derivation function based on Hashed message authentication codes (HMAC) is used, which can be used to take a source of initial keying material, and derive cryptographically strong keys from it [12].

Apart from the primitives given by the ciphersuite, EDHOC also utilize functions used for HMAC-based key derivation:

- **HKDF_extract(salt,IKM)**: The HKDF_extract function, uses the designated hash-function to extract a pseudo random key from a salt, and some input keying material [12].
- **HKDF_expand(PRK,info, length)**: expands a PRK to a desired length, and gives the option to provide arbitrary info to the function, which will be mixed in. Extract and expand are used together, to derive keying material. These functions are described in further detail at [12].
- **EDHOC-kdf(PRK, transcript_hash, label, context, length)**: Is a key derivation function defined through HKDF_expand. Where the extra parameters are truncated into a CBOR sequence, and given as info [22].

EDHOC message exchange

EDHOC consists of the Initiator (I) and the Responder (R), in LoRaRatchet terms, the Initiator is the ED, and the Responder would be the AS. EDHOC defines four messages, with the fourth one being optional, in the case that the initiator should have a stronger guarantee that the Responder has generated the same shared secrets. The initiator starts the protocol by sharing it's public ephemeral key, cipher suite and authentication method of choice. Following this, in message 2 and 3, the Initiator and Responder exchange hash values and ephemeral keys necessary for them to establish shared pseudo random keys. During the protocol, three pseudorandom keys (PRK) are calculated:

- **prk_2e**: Which is calculated by computing HKDF_extract on a empty bytestring as salt, and the shared secret between the ephemeral keys of both parties.
- **prk_3e2m**: Is calculated by running HKDF_extract on prk_2e as the salt, and the shared secret between the ephemeral key of I, and the static key of R.
- **prk_4x3m**: Is calculated by running HKDF_extract on prk_3e2m as the salt, and the shared secret between the static key of I, and the ephemeral key of R.

The latter is used to generate the final output. The final output of the EDHOC protocol, is the master secret, and master salt, which are used to derive shared application keys.

In the LoRaRatchet implementation of EDHOC, the fourth message is included, in order to ensure to the ED that the AS has generated the same shared secrets [8]. This ensures the Initiator, that the responder has agreed to it's identity.

EDHOC also support error messages, and if one party fails in their handling of a message, an error message is sent to the other party. They can then discontinue the handshake [22, p.37].

Message 1

We will now describe how EDHOC generates the four messages that it entails. For this section, we will describe the parties as I and R. Another thing to note, is that where EDHOC uses G_X, X and G_I, I to denote ephemeral and static keys of the initiator. We reference the ephemeral keys of the initiator PubEK_I, PrivEK_I, and PubSK_I, PrivSK_I for static keys. This lines up with the naming in our implementation, and has shown to be more idiomatic when showing the implementation to colleagues. The naming convention for the responder also follow the same pattern.

The EDHOC message 1, sent by the initiator is composed as follows:

- **Type**: An integer specifying whether signatures or static DH keys are used.
- **Suite:** A integer specifying the cipher suite suppported.
- **PubEK_I**: The public ephemeral key of I.
- **C_I**: A connection identifier chosen by I.
- **EAD_1**: External authentication data, which is optional to include or not.

Message 1 is a CBOR encoded sequence of the above mentioned items.

```
1 (Type, Suite, PubEK_I, C_I, ? EAD_1)
```

Listing 2.2: EDHOC message 1

Notice that the question mark at the end here, indicates that the EAD is optional, and may be omitted.

Message 2

For message 2, the responder constructs the following values, at this point, the responder is able to generate prk_2e and prk_3e2m since it has access to PubEK_R, and it's own static key.

- **EAD_2**: Optional external authorization data.
- **C_R**: A connection identifier that the responder chooses.
- **PubEK_R**: The public ephemeral key of R.
- **TH_2**: A transcript hash computed on message 1, the connection identifier, and the public ephemeral key of R.
- **ID_CRED_R**: A COSE header map that contains the key identifier (kid) that I can use to retrieve the static key of R.
- **CRED_R**: is the credential containing the public static key of R
- **MAC_2**: A mac value is calculated through EDHOC-kdf on the prk_3x2m, and using TH_2, and CRED_R and ID_CRED_R (and optionally EAD_2) as additional context.
- **P**: a plaintext is constructed by encoding ID_CRED_ID MAC_2, and optionally EAD_2 as a CBOR sequence.
- **KEYSTREAM_2**: A keystream is constructed through EDHOC-KDF, and derived through prk_2e, and TH_2, and expanded to the length of the plaintext.

The third message is composed by performing bitwise xor over KEYSTREAM_2 and p and getting a ciphertext_2. This ciphertext is concatenated with PubEK_R, and then encoded as a CBOR sequence with the connection identifier.

```
(PubEK_R || Ciphertext_2 , C_R)
```

Listing 2.3: EDHOC message 2

Message 3

At this point, the Initiator can retrieve the static key of the responder PubSK_R, through ID_CRED_R, and has access to the PubEK_R, and can therefore construct prk_4x3m.

- **EAD_3**: Optional external authorization data.
- **TH_3**: A transscript hash calculated on TH_2, and Ciphertext_2.
- **MAC_3**: A mac value is calculated through EDHOC-kdf on the prk_4x3m, and using TH_3, and CRED_R and ID_CRED_R (and optionally EAD_2) as additional context.
- **ID_CRED_I**: A COSE header map that contains the key identifier (kid) that R can used to retrieve the static key of I.
- **CRED_I**: is the credential containing the public static key of I.
- **K_3**: is a symmetric encryption key, derived through EDHOC_kdf, expanded on prk_3e2m, and TH_3, and expanded to the length of the given AEAD algorithms key.
- **IV_3**: Is an nonce, derived in the same manner as K_3, but expanded to the length of the IV in the given AEAD algorithm.
- **P**: The plaintext for message 3 is a CBOR sequence of ID_CRED_I, MAC_3, and optionally EAD_3.

The ciphertext for message 3, ciphertext_3, is computed by using the given AEAD algorithm with K_3 as the key, IV_3 as the nonce, P as the message to be encrypted, and TH_2 as associated data, which is encoded as a COSE enc_structure, Which is a CBOR array of a string identifier, a bitstring of a protected attribute, which in our case is a empty bitstring, and a protected attribute, which is TH_3[22, p.70].

Message 4

After the initiator has sent message 3, the responder can retrieve PubSK_I, of the initiator, and thus both parties can now derive prk_4x3m. However, it may be necessary for the initiator to receive confirmation that the responder has generated the same prk_4x3m, in this case, message 4 is sent.

- **EAD_4**: Optional external authorization data.
- **TH_4**: A transcript hash calculated on TH_3, and Ciphertext_3.
- **K_4**: is a symmetric encryption key, derived through EDHOC_kdf, expanded on prk_4x3m, and TH_4, and expanded to the length of the given AEAD algorithms key.
- **IV_4**: Is an nonce, derived in the same manner as K_4, but expanded to the length of the IV in the given AEAD algorithm.
- **P**: The plaintext of message 4 is a empty bytestring, or EAD_4, if it is used.

Finally, the Ciphertext_4 of message 4 is the AEAD encryption of P, encrypted with K_4, using IV_4 as a nonce, and a enc_structure containing TH_4 as associated data.

In figure 2.2, the messages as they are sent in EDHOC is displayed

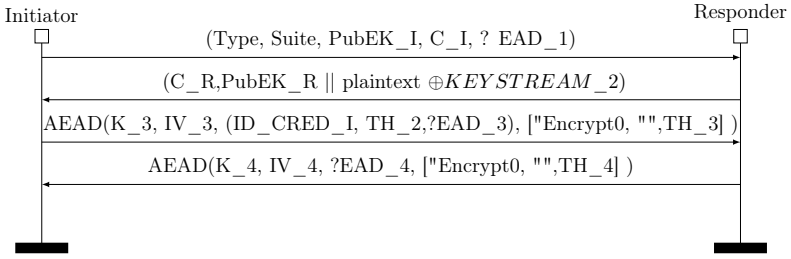

Figure 2.2: EDHOC messages outline with static DH authentication [22]

Completion

When both parties have generated the same prk_4x3m, they can now derive a common master secret, and master salt. These are made through EDHOC_kdf, and derived with the prk_4x3m and TH_4. Conventionally, EDHOC uses these values to establish the initial context for OSCORE. In LoRaRatchet, it should instead need to output initial keys for the Double ratcheting protocol, how this is done is specified in our chapter regarding the implementation.

2.6.1 Security properties of EDHOC

EDHOC is an extension of the family of SIGMA-I (SIGn-and-MAc) protocols, and inherits security properties from it. The SIGMA family of key-exchange protocols, describe a general manner of building authenticated key-exchange DH protocols. The "I" in SIGMA-I refers to the fact that the identity of the Initiator (the static key identifiers) is protected towards a active attacker, while the identity of the responder is only resistant to a passive attacker [22, p.41]. This makes sense in the domain of LoRaRatchet, where the identity of the server may be publicly known, while the identity of a single end device should be more protected.

EDHOC extends SIGMA-I, by including the integrity checks via MAC's and authenticated encryption, and authenticating on previous messages, which provides protection against replay attacks, and message injections [22, p.41].

EDHOC provides perfect forward secrecy of session keys between handshakes by using fresh ephemeral keys for every handshake, which makes the protocol resistant to the leakage of long-term keys [13, p.18] [22, p.41].

A formal analysis performed by Bruni et. al. [15] on a slightly older draft of the protocol, verified the perfect forward secrecy of the session key material, and mutual authentication properties. The analysis found that EDHOC ensures that once a handshake has been run, the responder and initiator are ensured that they agree the same session keys, each others identities and roles, and that this agreement is injective. Injective meaning that there is a one-to-one relationship between protocol runs. It is however important to note, that when using the mode of DH static keys to authenticate, the Initiator gets no guarantee that the Responder has agreed on the identity of the Initiator. This is however mitigated by including the fourth message, which ensures the Initiator that the same prk_4x3m has been reached for both parties.

A perk of using the static DH authentication mode, is also that both parties can deny having taken part in the protocol [22, p.43].

2.7 Double Ratcheting

As described, the LoRaRatchet protocol first establishes an security context with EDHOC. Once this is done, the double ratcheting protocol is employed. In LoRaRatchet, the output of the Join procedure is the initial receiving chain key (RCK) and sending chain key (SCK), and root key (RK). The Ratchet uses Key derivation functions to take some input, and output data, which should be indistinguishable from the input given.

In general terms, the Double ratcheting protocol ensures that two messages sent will never be encrypted under the same key. In order to achieve this, two different ratchet mechanisms are used, the outer, and the inner ratchet. The inner ratchet generates the encryption keys for messages, while the outer ratchet generates new contexts for the inner ratchet . The double ratcheting scheme ensures that two messages will never be encrypted under the same key, however it is also important that the scheme is resistant to lost, or out-of-order messages. The Double ratchet protocol employ mechanisms for this that will be explained in the following sections [17].

2.7.1 Symmetric ratchet/ Inner chain

The core ratchet is the symmetric ratchet, which uses a Key Derivation Function (KDF) to generate keys from a seed in a non-reversible way. For this purpose KDF functions are chained together to create KDF chains. In this context, the KDF functions are implemented using HKDF-extract and HKDF-expand. The KDF function takes a KDF key and some input, and then produces a key, which can be used to encrypt any application relevant data with authenticated encryption. The double ratchet mechanism consists of three KDF key types, the sending key, which functions as a KDF key input to the sending function; the receiving key, which is the same but for receiving messages, and lastly, the root key (RK), which is used in the root chain, described in the next section [17].

The RCK and SCK, (also called the chain keys) are used for the innermost ratchet, that generates message encryption keys (MK) that each individual message is encrypted with. Here, an RCK or SCK is given as input, together with some constant, which will yield a MK for that message. The KDF chains provide forward security because anyone who observes an output key or KDF key will not be able to derive the inputs that caused them. So this is the basic mechanism that ensures forward security of keys. This mechanism is pictured in figure 2.3.

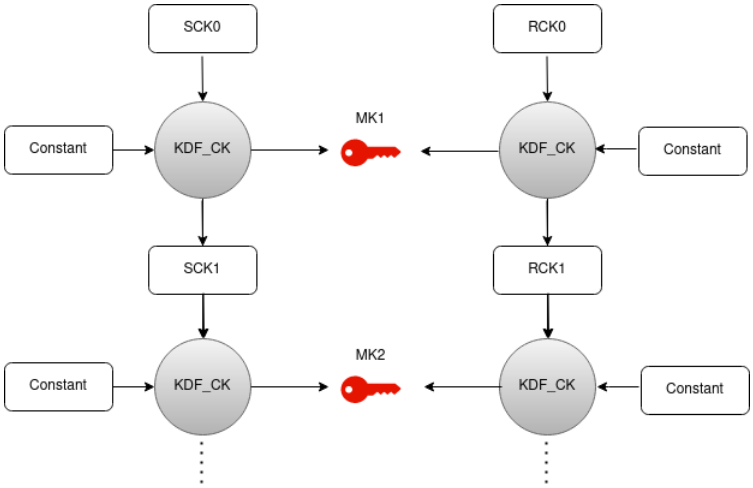

Figure 2.3: Inner ratcheting (Diffie hellman Ratchet) of the Double ratchet procedure

In the inner chain, a constant is used as input to the KDF chain. This is as specified by the Signal protocol. If the two parties run these chains in a synchronous manner, the receiving chain key of party ED will be the same key as the sending chain key of party AS. In this manner, the two parties can

derive the same keys, while permuting them.

2.7.2 Diffie Hellman Ratchet / Root chain

If the Double Ratchet procedure only consisted of this inner chain, then PCS would not be ensured since an adversary who learns the state of a device at one point in time would be able to decrypt future payloads for the next KDF round [17]. In order to ensure PCS, the Diffie hellman ratchet procedure (DHRP) is used. The DH ratchet ensures that the SCK and RCK are updated, every time the communicating parties switch roles, in terms of being either the sender or the receiver of messages. This behaviour is referred to as a asynchronous ping-pong behaviour. In the standard double ratcheting protocol, each of the parties are initialized with a identical secret key (SK), and one party holding the ephemeral public key of the other [17]. The initiation, and a example of the DHRP procedure when run in the ping-pong manner is shown in figure 2.4.

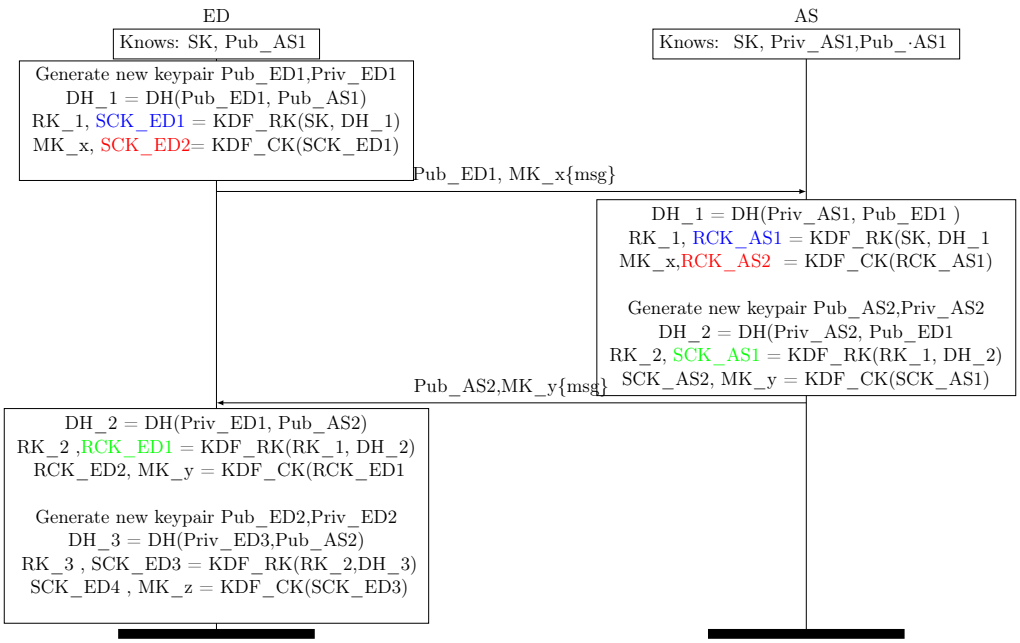

Figure 2.4: Double ratchet "ping pong" behaviour

1. The ED starts by generating a ephemeral keypair and generates a shared secret. The shared secret and the SK is given as input to a KDF function, providing a first SCK and RK. The sending chain is then advanced once, yielding a new SCK, and MK. A message is encrypted under the MK, and sent along with the new public key.

2. The AS receives the message, and advances it's receiving chain in the same manner that the ED advanced it's sending chain. A ephemeral keypair is generated, and a shared secret is generated from the incoming public key, and newly generated secret key. The shared secret is used to advance the root chain, and then derive a new MK and SCK via the sending chain. This concludes the DHRP step for the AS.

3. Lastly, the ED receives the new public key, and encrypted message, and can thus advance it's receiving chain to obtain the needed MK for the incoming message. A new ephemeral keypair can now be generated, the root chain can be advanced, and the new SCK and MK generated.

LoRaRatchet root chain

The ping-pong behaviour of the Double Ratcheting protocol is not adopted in LoRaRatchet. Instead, the DHRP is only set to happen at certain preset intervals, called the DHRP interval limit. This is done to save the cost of running the procedure. In LoRaRatchet, the EDHOC join procedure, directly outputs a RCK, SCK and RK, allowing the two parties two encrypt sent messages without doing a initial DHRP.

A DHRP is instead initiated by including a counter in all messages, indicating how many messages are sent under the current DHR context. If the counter reaches the set interval limit, it will indicate to the parties to reset the security context with a new DHRP. Since each party holds a Fcnt counter, this value is used as this counter. When the DHRP has been finalized, the FCnt can be reset to 0. Since the Fcnt is a two-byte value, it sets a maximal DHRP interval at 65.535 messages.

This deviation from the regular Signal protocol weakens the Post compromise security of the protocol, since an attacker that could obtain a single decryption key, could be able to obtain future messages, as long as a new DHRP is not performed.

With this change to the protocol, the asynchronous ping-pong behaviour can no longer be held, and another procedure has to be chosen.The LoRaRatchet protocol suggest two different methods for achieving this variable DHRP interval. The suggestion for a DHRP looks as following:

- **1**: The AS receives an uplink message, registers that the FCnt has reached the set limit, and initializes a new DHRP by generating a new ephemeral keypair, and saves those keys. The AS prepares a message HRReq (DHR Request) by packing this the public key with a two byte DHRReqNonce counter, indicating how many DHRReq's have previously been sent.
- **2** The ED receives the DHRreq, and checks that the DHRReqNonce is larger than any DHR-ReqNonce previously received. Then, it generates a new ephemeral keypair, and constructs a DHRRes (DHR Response) message containing the new public key and a two byte nonce called DHResNonce, which keeps track of how many DHRack messages have been sent.
- **3**: The AS receives the DHRRes, validates the associated nonce, performs a DHR ratcheting step with the new keys, which means resetting FCnt counters and chain keys. From this it constructs a DHRack (DHR acknowledgement) message, encrypted under the previous context.
- **4**: Lastly, the ED receives the DHRack message, and updates it's context. This concludes the DHRP, as both parties now share the same new context. And the ED can now send uplinks under the new context.

[8, p.40]

A less complex alternative is also suggested in the LoRaRatchet report. This approach lets the ED initiate the DHRP, and only consist of two messages:

- **1**: As the ED is about to send another message, it registers that the set limit of messages has been reached by viewing the FCntUp. It then generates a new keypair and pairs the public key with a DHRReqNonce (DHR Nonce), to create a DHRreq message that it sends. The newly generated keys are stored, and the ED keeps it's old security context
- **2**: The AS receives the DHRReq message, and validates the DHRReqNonce. Upon this, it generates a new keypair, and updates it's own security context with the newly generated keys. The AS then constructs a DHRAck message with it's new public key, and a counting DHRAckNonce. This message is sent to the ED, who can then update it's context.

[8, p.68].

This latter option simplifies the protocol to only two messages, and it shifts the responsibility of the initiation to the ED. This may be preferable, as it gives the ED more control over when to ratchet. It is important to note about this latter option, is that it makes a few assumptions about the nature of the communication. Since the decision of whether or not to perform a DHRP step is decided by the ED evalutating upon it's FCntUp value, downlink messages do not have an effect on when a DHRP will happen. This is however consistent with most LoRaWAN use cases, where the vast majority of messages are uplink. The Things Network (TTN) who provide one of the most prevalent LoRaWAN ecosystems, recommend restricting downlink messages to a maximum of 10 messages per day [29]. This report will for this reason consider a case where the vast majority of messages are uplink.

In order to reuse primitives from the EDHOC handshake, the x25519 is reused for key generation, and AES-CCM-128 is reused for authenticated encryption, and Sha256 for key derivation.

2.7.3 Double Ratchet lost message handling

In order to deal with lost or out order messages, and to correlate messages, the Double ratcheting of Signal specify a header that should be carried along all messages. This header contains a value which for our purpose is called the FCnt, indicating how many messages have been encrypted under the current DHR contexts sending chain, and a value pn, which indicates how many messages were encrypted in, the previous DHR context. Importantly, it also sends along the public key under which the corresponding MK was generated, as explained in the previous section, which in the signal protocol, is used to indicate that new DHR contexts should be created[17].

Lost messages are to be stored in a Dictionary/Hashmap like datastructure, where the identifying key is a tuple of the used public key, and the FCnt counter for that message.

Lost messages within a single sending/receiving chain are dealt with by letting the receiving party check whether the current FCnt value they received is larger than the last one they received. If so, the receiving party is allowed to advance their receiving chain, until the FCnt value matches, and store all the skipped MK's, which can be used for older messages, if they are to show up at a later time.

Another case of handling lost messages, is the case where a new DHRP process is initiated, and there still may be lost messages in the previous DHRP context. In the Signal protocol, this is where the pn value becomes relevant. If a header arrives containing a new public key, this indicates that a new DHRP procedure should happen. The receiving party will then evaluate on the pn value, and skip the amount of messages from the previous DHRP context, that has not yet been decrypted [17].

In LoRaRatchet, the sending of the 32-byte public key along every message is omitted, due to the space requirements of the PHYPayload. A value corresponding to the pn value is also omitted. The solutions for these issues are detailed in our section 4.3.1 regarding the implementation of the Double ratchet.

2.7.4 Double Ratchet security properties

The Double Ratcheting protocol provides various properties. Most notably is the FS and PCS.

FS is achieved through the symmetric chain, which provides forward secrecy to each message individually, meaning that it is not possible for an attacker to derive old keys from disclosed keys in the future. Or in other words, it's not possible to decrypt old messages, if current session keys have been disclosed [17].

PCS is achieved via the Diffie Hellman Ratchet by mixing in the DH shared secrets into the root keys used to generate the symmetric ratchet. This utilizes the property of break-in recovery of KDF chains, stating that future output keys should appear random to adversaries who have learned the input keys, as long as enough entropy is given. This ensures that even if an attacker gains access to a session key, future session keys are kept confidential. In the LoRaRatchet adaption of the protocol, the variable

length DHRP interval means that this property can be weakened, depending on how the interval is chosen [8] [17].

2.8 LoRaRatchet experiments findings

The LoRaRatchet report evaluates their protocol by constructing a model of the protocol, using the PRISM model checker. Here, different scenarios are modelled, simulating the power usage of an ED across three different Data rates, sleep periods and DHRP intervals. Since the introduction of the variable interval DHRP, is the biggest deviation from standard protocols that the LoRaRatchet protocol has, the most interesting observations is the impact of the DHRP interval on power consumption. The power consumption is measured as the long-term average (mJ) consumption of a ED, by the second, from which a battery lifetime estimate is calculated, based on a 2200 mAh battery.

The effect of the DHRP interval, is best shown when the sleep period is shorter (30 minutes). In table 2.3, the estimated battery lifetime in years is shown across DHRP intervals. For ease of reading 2.4, the relative power consumption between DHRP intervals is listed across different transmission success rates (TSR). The TSR imitates the case where a ED is deployed in sub-optimal environments, and a certain amount of transmissions must be assumed to fail. The authors were unable to measure a DHRP interval of 256 and 1024, and thus these are omitted for the 100% success rate case.[8, p.60]

DHRP interval /TSR	**1**	**4**	**16**	**64**	**256**	**1024**
100%	2.81386	2.87866	2.92923	2.94223	-	-
80%	1.94233	2.13608	2.41485	2.50765	2.53278	2.53919
60%	0.72577	0.78890	0.96167	1.04427	1.06963	1.07632

Table 2.3: DHRP interval, 2200 mAh battery lifetime estimate in years(30 minutes sleep) [8, p.97]

DHRP interval /TSR	**1-4**	**4-16**	**16-64**	**64-256**	**256-1024**	**Total increase**
100%	2.3%	1.7%	0.4%	-	-	4.4%
80%	9.1%	11.5%	3.7%	1.0%	0.3%	23.5%
60%	8%	18%	7.9%	2.4%	0.6%	32.6%

Table 2.4: DHRP interval, relative increase (30 minutes sleep) [8, p.60]

The modelling finds that in the case of a 100% successful message transmissions, there is around 4.4% difference between the highest interval, and 1. While this is significant, the differences are far more pronounced when taking lower TSR's into account. In the worst case with 60% successful messages transmissions, the total increase between the interval of 1 and 1024 is 32.6%.

The difference between the cases, is attributes to lost DHRP messages, which will force the ED to listen for longer periods of time, and resend the DHRP messages [8, p.60]. Furthermore, this effect is compounded when the DHRP interval is lower, since the net amount of messages transmitted is increased, and thus also lost messages. For this reason, the LoRaRatchet authors suggest that careful considerations should be made when deciding on how to set the DHRP interval value, which should factor in the longer-lived sessions given by a higher interval, against the higher power consumption which a lower interval brings.

2.9 Rust

Due to the properties detailed in the following section, Rust has been chosen as the implementation language of this protocol. A more extensive explanation of the languages features can be found in the Rust book [11].

Rust is a statically typed language with a robust type system enforced at compile time. Instead of having a garbage collector, Rust implements an ownership model, which governs how variables can be accessed and borrowed. This means that the developer will be notified at compile-time if invalid memory access occurs. All variables are owned and can only have one owner, and whenever it goes out of scope, it is deallocated. The absence of a garbage collector means that rust is able to achieve faster runtimes.

By default, variables in rust are declared with a **let** keyword. By default, variables are immutable, but can be made mutable with a **mut** keyword. If a created variable is passed to a function, the function will take ownership over it, however if it is not necessary for the function to own the value, if it only is reading it, the variable can be borrowed. A borrow is done by prefixing the value with **&**, which creates a reference to the variable, which can allow several functions to borrow it. It is also possible to create mutable references by prefixing the variable with **&mut**, which allow references to be mutated by functions that do not own them, with the restriction that there only can exist a single mutable reference to each variable at a time.

```
1     fn ratchet_decrypt(&mut self, phypayload: Vec<u8>)
2     -> Result<Vec<u8>, &'static str> {
3         let deserial_phy =  deserialize(&phypayload_phy)?;
```

Listing 2.4: Generating root key from master secret and salt

An example from our own implementation of the Double ratchet protocol, that encapsulates a few of the beforementioned features is shown in listing 2.4. Here, a mutable reference to a **self** object is taken as argument, which references the structure for which the function is implemented for, but allows changes to the structure to be made, as long as only one such mutable reference exists. The other argument **phypayload** is a vector of bytes which is owned by the function, disallowing the caller to reuse the given argument again. The **phypayload** is then given as a borrowed reference to the deserialization function, that only need to read it.

The function returns a **Result**, which is a enum, which either wraps an **Ok** value of a vector of bytes, or an error, which here is represented with a string literal. The **deserialize_phy** call is ended with a **?** operator, meaning that any error happening in that function will be propagated to the caller.

Using this ownership model, Rust is able to provide us with a running time that is comparable to c, but without needing memory management (apart from understanding the ownership model), and without the added overhead of running a garbage collector like in a language such as Java.

Compared to C, Rust's ownership model can guarantee memory safety, and thus avoids many of the risks that is often associated with writing C programs [11]

Rust supports integration with c libraries via foreign function interfaces (ffi). The justification for choosing Rust over a language like c, which is also often used in embedded systems, is the comparable performance features, as well as the safety features that Rust has, which makes it a common choice for developing embedded devices.

When writing rust for embedded environments, the environment can either be a hosted environment, with access to the Rust standard library, usage of a heap, allowing dynamic allocation. Rust code can also run in what is called "Bare metal" environments, where the a standard library can not be loaded, and there is no access to heap. Rust projects made to operate in the latter can be tagged with the `[#![no_std]]` attribute, indicating that the project does not need access to the standard library. Projects with this attribute will instead have access to a core library with a subset of the standard features [11].

3 Hardware

This section will mention the hardware we worked with, in order to perform the experiments of this report.

3.1 SX1276

In order to enable LoRa communication, we acquired a SX1276 breakout board. The SX1276 is developed by semtech, and is a fairly standard lora module which comes at a low cost, and can provide the typical properties of LoRa, with long ranges and low power consumption [26].

There exists a platform-agnostic rust driver for the SX1276, which provides a API for polling and transmitting Radio PHY structures [31]. The device can with the driver be connected to a microcontroller through a serial peripheral interface (SPI).

The device can be set to operate in a sleeping mode, drawing very low amounts of power between 0.2 and 1 microamperes (uA).When needed, the device is set in it's receiving mode, which ups the power consumption to a range between 10-12 milliamperes (mA). Transmitting data is far more costly procedure, and is estimated to set the current at 120 mA [26, p.14-19]

An important note, is that the driver repository had not been given regular maintenance, and it was necessary to create our own fork, introducing a way to reset the chip registers, which needs to be done as the sender object is created, or configured. A pull request has been opened, but the original authors have been unresponsive. The edited fork can be found at [3].

3.2 ESP32

For some of the experiments conducted in this report, we have used the ESP32 microcontroller. The ESP32 is a 32-bit micro-controller built by Espressif Systems It is built to be low-cost and having low power requirements, but include many features, such as built-in wifi and Bluetooth modules and 512 Kb of RAM, and as micro-usb port that can be used for powering the device, or flashing onto it. ESP32's can be used in many differing domains, such as small wearable devices, or thermostats [14]. The ESP32 that we acquired in particular was the ESP32-wroom, equipped with a xtensa 32-bit cpu.

In order to compile Rust projects on the ESP, we installed the Rust Espressif compiler toolchain, that uses a fork of the standard rust compiler [19]. Espressif provide their own development framework written in c (ESP_IDF), and provides some bindings for Rust usage.

For the purposes of this report, it was necessary to measure the power consumption of the device while running our implementation. We acquired a UM34C usb power meter, with the ability to provide voltage readings, and power capacity readings in mAh down to 0.001 ampere. The device features a male, and female usb port, and a lcd screen for displaying readings. We can use this device to run the protocol on the ESP32, and read the power consumption in mAh over time [9].

The experiments on the ESP32 were originally intended to communicate through the Lora module. However after much research, and consulting with the ESP32 ESP_IDF rust community, it became clear that the Serial Peripheral Interface binding provided by ESP_IDF, was incompatible with the SX1276 driver. This has to do with both the SPI of ESP_IDF, and the SX1276 driver needing to take control of the chip select pin, which is used to connect I/O pins to the internal circuitry of the device. This disallowed us from actually using that pin in our setup, which took the SX1276 module on the ESP32 off the table as an option.

3.3 Raspberry Pi

Due to issues presented with the ESP32 and our lora module. this report also features an implementation on raspberry pi's.

We used two raspberry Pi 3B+ single board computer, as we already had them on hand. The important specs are as follows, the raspberry pi is equipped with a quad core 64bit ARMv8 processor running the linux distro Raspbian OS, has 1GB of ram and access to a 40-pin header [18]. This gives us the possibility to extend the functionality of our raspberry pi's, and thus add a module such as the semtech SX1276 lora chip, which gives us the capabilities to communicate over lora for an otherwise non lora device.

The GPIO headers on the raspberry pi supports 3.3v and 5v, and gives us access to several SPI interfaces, which we need 1 of for the SX1276 chip [26]. This enables us to wire up and transmit and receive by using the SX1276 module. The raspberry pi is also able to power the SX1276 module.

Since the raspberry pi's are running on a ARM processor we needed to have a ARM based build chain, for this we utilized a utility called cross [5] which allow cross-compilation of Rust for dockerized builds.

Furthermore the raspberry pi is running a fully fledged Linux based operating system which defeats some of the purposes regarding having embedded devices, since the Raspberry Pi cannot be classified as a constrained device, and thus measuring it's total consumption would not render useful data.

This is however mitigated by us being able to measure the consumption of the SX1276 module itself. This is to get an idea for the cost of sending transmissions themselves, and how many transmissions are sent. In order to do these measurements, we were able to aquire a Otii Arc, which is a power analyzer, power supply and log sync, all included in one device. While having the lora module wired up to the raspberry pi, we can power it with a 3.3v connection from the Otii Arc, which is connected to a computer with the assoicated Otii Arc desktop application running on it [16]. This way we can view and analyse the power draw of the LoRa module over time.

4 Implementation

This section will specify the steps in the implementation of the two sub-protocols of LoRaRatchet, EDHOC, and the Double ratcheting protocol. Both libraries are I/O free, and it is up to the caller to provide the source of randomness. Both libraries also implement the `#![no_std]` attribute, but do need access to an allocator.

4.1 Messages and Omitted fields

The PHYPayload composition of LoRaRatchet as described in section 2.5.1, is based on the one of LoRaWAN.

Since the implementation of this report considers on a implementation working directly on LoRa, there are certain fields of the LoRaRatchet PHYPayload that are omitted, as they either are optional, or primarily used in the LoRaWAN infrastructure.

In the initial Mtype field field a field of 0 to 3 bits are allocated for being reserved for later use (RFU). This field is omitted, and instead the Mtype fills those bits.

Two omitted fields are *Fport* and *FCOpts*, which are defined as optional fields in the LoRaWAN specification[23, p.18]. Another omitted field is the *Fctrl*, which consist of 1 byte of bits indicating configuration options regarding the device and the optimal data rate. As this report is an enclosed experiment, where we consider two devices with knowledge of each others settings, this field is omitted [24, p.19].

In the following implementation of the EDHOC messages, the reader will also notice that the Join procedure messages of the protocol do not carry the nonce, or the mic tag, that are specified in section 2.5.1. This is because the EDHOC messages do not include any use for nonces that are transmitted along the messages, and only include authentication fields that are contained within the EDHOC messages themselves. EDHOC also does not include explicit mic tags, other than the authenticated data that is included in the AES-CCM ciphertext of message 3 and 4.

Each message carries a MType, indicating the type of the message. The Mtypes of each message are as shown in table 4.1.

Type	**Payload**	**Usage**
0	M1	EDHOC
1	M3	EDHOC
2	M3	EDHOC
3	M4	EDHOC
4	Rejoin request	DHRP
5	DHR request	DHRP
6	DHR Ack	DHRP
7	Uplink	Application message
8	Downlink	Application message

Table 4.1: Implementation message types

4.2 EDHOC implementation

In order to implement the Join procedure part of the LoRaRatchet protocol, we started from an already existing implementation of EDHOC, which can be found at [6]. This version was however heavily deprecated, and was based on EDHOC from 2019 [21], since then EDHOC has received 15

updates. Apart from this, the already existing implementation of EHDOC relied on authentication with raw public keys, which in our implementation needs to be static DH keys.

In order to update this repository, the outwards facing EHODC API had to be rewritten to comply with four messages, using static DH keys, allow for external authorization data to be sent, define the EDHOC_kdf function and allow it to generate the necessary values such as MAC values, use bitwise XOR for generating the second ciphertext, instead of encrypting it, introducing PRK values, instead of relying directly on a single DH shared secret etc. Additionally, there are several smaller changes, such as the values and messages being constructed differently, which we will not delve further into.

The primitives and functions of the library has been validated up against vectors provided by the EDHOC library

For this purpose, we wanted a general purpose API, that could both be utilized in the use case of this project, but also serve as an addition to a more updated implementation of EDHOC, which there exist very few of.

One place where this implementation differs from otherwise normal implementations of EDHOC, is the output of the handshake. Conventionally, EDHOC uses EDHOC-KDF to derive a master_salt, and a master_secret. In this implementation, these values are still generated, but then HDKF-expand is used to create a RCK, SCK, and a RK, where the RCK of the ED is the same as the SCK of the AS, and vice versa. This is in order to be able to provide an initial context for the double ratcheting protocol.

4.2.1 EDHOC API

The EDHOC implementation is represented to the caller by a API, which defines the two roles, Party I, and Party R (ED and AS). Each party have certain operations associated with them, and is only allowed to perform those steps in specific order of operation. Rust's ownership model offers some useful functionality here, since it is possible to represent the different states of a party, in a manner that only allows the party to enter a certain state, if it has done the necessary steps.

The overall structure for Party I can be seen in listing 4.1. The PartyI struct represents the initiator itself, and simply wraps a trait of type **PartyIState**. A trait is a abstract definition, similar to a Java interface, which we can use to define the behaviour of types.

```
pub struct PartyI<S: PartyIState>(pub S);

pub trait PartyIState {}
impl PartyIState for Msg1Sender {}
impl PartyIState for Msg2Receiver {}
impl PartyIState for Msg2Verifier {}
impl PartyIState for Msg3Sender {}
impl PartyIState for Msg4ReceiveVerify {}
```

Listing 4.1: State struct for party I

The abstract trait is then implemented, for all the states that the initiator needs to be able to arrive at. These states are represented as structs, that contain all the needed information for the party to do the next step. The actual contents of the state structs are defined elsewhere.

```
pub struct Msg1Sender {
    ead_1: Option<Vec<u8>>,
    c_i : Vec<u8>,
    priv_ek_i: StaticSecret,
    pub_ek_i: PublicKey,
    pub_st_i: PublicKey,
```

```
    priv_st_i: StaticSecret,
    kid_i: Vec<u8>,
}

impl PartyI<Msg1Sender> {

    pub fn new(
        c_i: Vec<u8>,
        ead_1: Option<Vec<u8>>,
        ephemeral_secret: [u8; 32],
        priv_st_i: StaticSecret,
        pub_st_i: PublicKey,
        kid_i: Vec<u8>,
    ) -> PartyI<Msg1Sender> {
        ... create keys from ecdh_secret ...
         PartyI(Msg1Sender {
            ead_1,
            c_i,
            priv_ek_i,
            pub_ek_i,
            pub_st_i,
            priv_st_i,
            kid_i,
        })
    }
    pub fn generate_message_1(
        self,
        method: u8,
        suites: u8,
    ) -> Result<(Vec<u8>, PartyI<Msg2Receiver>), EarlyError> {
        ... Prepare and serialize message 1, and construct Msg2Receiver  ...
    }
}
```

Listing 4.2: State struct for party I

In listing 4.2, we see how the initial **Msg1Sender** state for I is constructed, passing arguments such as the connection identifier, ephemeral keying material, static keys, and the kid parameter, which is the identifier of the initiators static key. The optional external authorization data is passed as an option, which may be a None value, in the case that it is not used.

After calling the constructor, which generates a Msg1Sender object, the caller can then use it to generate a message 1 via the **generate_message_1** function, which returns the bytes of message 1, and a new state **Msg2Receiver**, which is used to receive the second message. **generate_message_1** takes self as an argument, which means that the function needs to be called on a instance of **Msg1Sender**, and that the object will be owned that function, which means that it cannot be used to generate another message again.

```

impl PartyI<Msg2Receiver> {
    pub fn unpack_message_2_return_kid_ead(
        self,
        msg_2: Vec<u8>,
    ) -> Result<(Vec<u8>, Vec<u8>, Option<Vec<u8>>,PartyI<Msg2Verifier>),
    OwnOrPeerError> {
    .. Deserialize message ...
```

```
    ... output kid, connection identifier,
        external auhorization data and Msg2Verifier ...
    }

    pub fn unpack_message_2_return_kid(
        self,
        msg_2: Vec<u8>,
    ) -> Result<(Vec<u8>, Vec<u8>,PartyI<Msg2Verifier>), OwnOrPeerError> {
        ... Deserialize message ...
        ... output kid, connection identifier, and Msg2Verifier ...
    }
}
```

Listing 4.3: State for receiving message 2

After generating message 1, the **Msg2Receiver** can then be used to call **unpack_message_2_return_kid** or **unpack_message_2_return_kid_ead**, depending on whether or not external authorization data is expected. These functions consume the given **Msg2Verifier**, deserializes the incoming message 2, and extracts the kid, connection identifier, and optionally **ead_2**.

The only way for the caller to be able to unpack a message 2 is if the preleading steps have been followed, and it is also not possible to verify a message, unless it has first been unpacked. This means that this structure leverages Rusts ownership model to disallow users to unpack more than one message in one protocol run, or unpacking a message 2, before being initialized and generating message 1. This way, the API safeguards against misuse.

For error handling in the EDHOC api, the errors **OwnError** and **PeerError** are used. An **OwnError** indicates the occurrence of some error happening locally, such as a failed integrity check, or a bad decryption. These errors should be handled by generating some error message and sending it to the other party, before aborting the protocol. A **PeerError** models the case that an error was received in one of the incoming messages, and thus generated by the other party, in case of a **PeerError**, the protocol is just stopped. For the very first message, an Error called **EarlyError**, which models the case that something fails while generating the first message. In this case, no message needs to be sent, and the protocol can simply discontinue.

Lastly, the EDHOC protocol usually outputs a master secret and a master salt. For our use case, it is instead needed that a RK, RCK and SCK is output, to serve as an initial context for the Double ratchet protocol. This is achieved by passing master secret and the master salt to a function that computes HKDF_expand on a PRK generated from the secret and the salt, and including a constant, ensuring that the output values are different.

```
        let rk = util::extract_expand(
            &self.0.master_secret,
            &self.0.master_salt,
            "RK0",
            32,
            )?;
```

Listing 4.4: Generating root key from master secret and salt

In listing 4.4, it is demonstrated how the root key is created, by running HKDF_expand on the master secret, and salt. The RCK and SCK are generated in the same way, but with different constant strings passed to them.

4.2.2 EDHOC in LoRaRatchet

In this section, we will specify how the LoRaRatchet protocol uses EDHOC, and how the EDHOC messages will look when dressed in a PhyPayload. One important note, that may cause confusion, is that we will refer to the two parties as the ED and the AS, rather than Initiator and Responder.

The DEVEUI, and the APPEUI, are the 8 byte IEEE EUI64 adresses, that identify the ED and the AS. In the first message, the DEVEUI is passed as the connection identifier, and the APPEUI is passed as external authorization data, such that the AS can verify that the message has reached the correct server. A full model of a successful EDHOC join procedure as it is implemented in LoRaRatchet can be seen in figure 4.1.

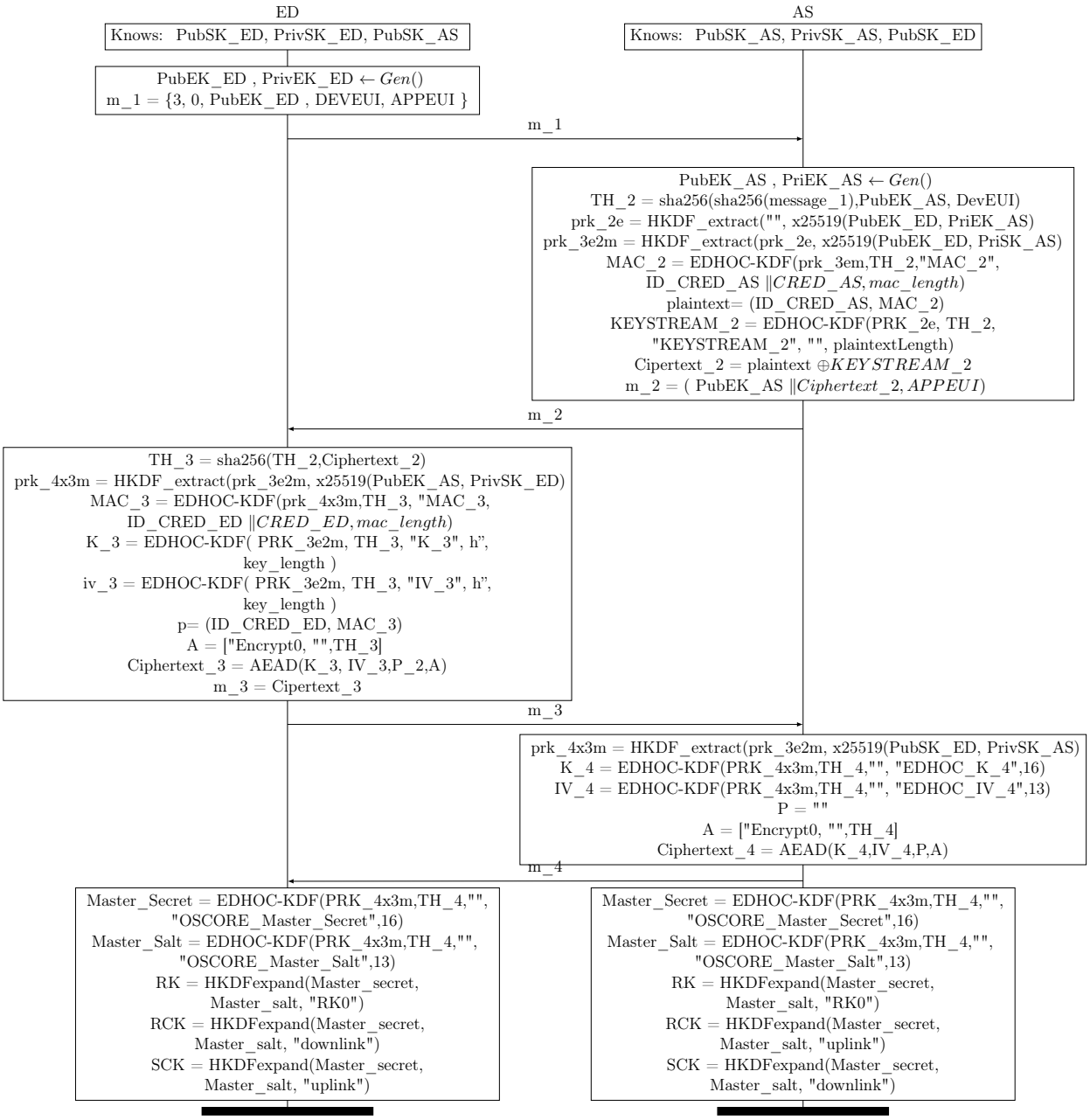

Figure 4.1: LoRaRatchet Join procedure [22]

The handshake modelled in figure 4.1, shows how the EDHOC messages in the LoRaRatchet Join

procedure are constructed, which in terms of the PHypayload, these messages are the FRMPayload. This means that the figure does not account for the additional PhyPayload fields. In table 4.2, the PHYPayload of message 1 is displayed. The DEVEUI is passed as the connection identifier, and the APPEUI is passed, using the option of external authorization data.

Field/bytes	Mtype /1	FCnt/ 2	SUITE /1	METHOD /1	ED ephemeral pk /32	DEVEUI /8	APPEUI /8
			CBOR encoded sequence (56 bytes)				

Table 4.2: Message 1 composition (59 bytes)

In the first message, a Mtype and frame counter is appended to the FRMPayload. Apart from the omitted fields mentioned in section 4.1, the EDHOC PHYPayloads also do not carry any 13-byte nonce. This is because the EDHOC messages have no use for such a nonce. When message 1 has been serialized, and the additional fields have been added, the PHYPayload has a length of 59 bytes, leaving room for 5 bytes of additional field, in the case that may become relevant.

For the second message, the PHYPayload of which can be seen in figure 4.3, the AS sends back the APPEUI as a connection identifier, and generates a DevAddr, which is sent as a field in the PHYPayload, and then used as an identifier for all future communication [8, p.84].

Field/bytes	Mtype /1	FCnt/ 2	Devaddr /4	APPEUI /8	AS ephemeral pk /32	ciphertext /11
				CBOR encoded array (54 bytes)		

Table 4.3: Message 2 composition (61 bytes)

Including the fields appended by the communications layer, this message takes up 61 bytes, leaving 3 btyes of room for additional fields.

The third message, is the first message encrypted with AES-CCM, so the FRMPayload simply consist of the encryption of MAC_3 and the key retrieval credential of the ED.

Field/bytes	Mtype /1	FCnt /2	Devaddr /4	Ciphertext_3 /19
				CBOR encoded sequence (20 bytes)

Table 4.4: Message 3 composition (25 bytes)

The fourth message simply consist of the AES-CCM encryption of the empty string. For that reason, the ciphertext only consist of the implicit 8-byte mac tag, encoded as a CBOR sequence.

Field/bytes	Mtype /1	FCnt/ 2	Devaddr /4	Ciphertext_4 /8
				CBOR encoded sequence (9 bytes)

Table 4.5: Message 4 composition (15 bytes)

These four messages make up the EDHOC messages of LoRaRatchet. The CBOR sequences is specifically the EDHOC messages, while the remaining fields are part of the PHYPayload.

4.3 Double Ratchet implementation

This section will specify the implementation of the Double ratcheting algorithm, as specified by the LoRaRatchet protocol. The inspiration for the implementation is based on the examples provided at

the signal specification [17]. As mentioned, the introduction of a variable interval for how often the DHRP interval should happen, is the biggest deviation from the standard double ratcheting protocol. This introduces a couple of design decisions to be made regarding how lost messages are handled, and how message correlations are made.

The output of the EDHOC Join procedure is an initial RK, SCK and RCK. In the regular signal protocol, it is required that the initialization is started of with a exchange of public keys to initialize the first ratchet, together with the secret key output of the 3XDH handshake. In LoRaRatchet, the EDHOC handshake directly outputs the material necessary to directly derive the same encryption keys.

4.3.1 Double ratchet message composition

The implementation of the Double ratcheting protocol is made specifically for the LoRaRatchet setting. This means that the library itself handles the entire PHYPayload, since all the fields of the PHYPayload should be authenticated by the protocol. The double ratchet procedure handles the long-term communication of a ED to the AS, and thus consist of four different messages with each their Mtype. The ED sends uplink messages with Mtype 7, and DHRP requests with Mtype 5. The AS sends downlinks of Mtype 8, and DHRP acknowledgement message with Mtype 6. The PHYPayload sent during the double ratchet procedure contain the already defined PHYPayload fields carried by the Join procedure messages. A 13-byte nonce is also included, used for the AES-CCM encryption of payloads. Through the encryption, a implicit mic field of 8 bytes is also in the PHYPayload, this mic value is carried in the FRMPayload, which consist of a AES-CCM ciphertext of whatever application data is sent.

In the double ratcheting of signal, a 32-byte public key is sent along every message. As mentioned in section 2.7.3, this key is far too large to include in every PHYpayload. However, the Double Ratcheting protocol still needs some way for the other party to know under which public keys, the message was encrypted under. In order to circumvent this, another field was added to the PHYPayload. This field is the DH_ID, of two bytes, which for every PHYPayload indicate which DHR context it was encrypted under, and is incremented for every successful DHRP.

Field	Mtype	Nonce	Devaddr	FCnt	DH_ID	FRMPayload/ Ciphertext
Bytes	1	13	4	2	2	0...x

Table 4.6: Implementation PHYPayload composition

In regular uplink and downlink messages, the ciphertext, simply consist of the AES-CCM ciphertext of the data to be transferred, and authenticated on all the other fields of the PHYPayload. Including the implicit 8-byte mic tag in the ciphertext. This allows the application data to be at most 34 bytes, or 32 if Fport and Fctrl are to be included. There could also be room for a FOpts field depending on the size of the FRMPayload.

The DHRreq, and DHRack messages also reside in the encrypted FRMpayload, and as described, they consist of a new 32-byte public key, and a DHRAckNone, or DHRReqNonce of two bytes. These nonces are used to correlate the DHRP messages, so that old DHRAck's or DHRReq's are not injected. Since the DHR messages themselves take up 34 bytes, this sets the PHYPayload at 64 bytes for those messages, and leave no room for additional LoRAWAN fields. It also does not leave any space for a field that could correspond to the pn field of the signal protocol. In terms of the size of the PHYPayload, these messages are the most problematic to realize. Since the number of encrypted messages in the previous DHRP context is not encoded in the messages, another solution had to be found, for how to allow to the protocol to be resistant from message loss in previous DHRP contexts. The solution became to let either party skip a constant amount of keys every time they perform a DHRP. This allows the protocol to still be resistant to lost messages across DHRP sessions, without having the

PHYPayloads carry a pn value. For the purpose of this implementation, this constant value is set to 10, but the decision of this number may depend on the conditions the product is to be deployed in.

4.3.2 Revised DHRP procedure

In section 2.7.2, two DHRP operations are described. As proposed by the LoRaRatchet report, the DHRP procedure with two messages was opted for [8, p.68], this procedure is simpler than the four-message alternative, and let's the ED dictate the DHRP solely. While implementing this feature, an issue was uncovered regarding this approach. While the ED waits with updating it's security context until a DHRAck message has been received, the AS updates it's context as soon as a DHRReq is received. In the case that the DHRAck message is lost in transmission, the AS will have entered the new context, but the ED will be stuck in it's former state.

For this reason, we implemented a slight change to the operation of the procedure. In this procedure, the DHRP consist of the same two DHRReq and DHRAck messages as described earlier. But in order to be more resistant to dropped messages, the AS should only update it's context when it receives the next uplink message after the handshake has been made.

- **1**: The ED registers that the FCntUp has reached the interval DHR limit. It then generates the new keypair, and increments its DHRReqnonce, and prepares a message containing the public key and the DHRReqnonce. This message is encrypted under the sending chain and sent.
- **2**: The AS receives the DHRReq message, and decrypts with it's own receiving chain. The AS can then generates it's own new keypair, and constructs a DH shared secret with the incoming public key of the ED, the shared secret is stored. The AS increments its DHRAcknonce, and prepares a DHRack message, which is encrypted under its sending chain.
- **3**: The ED receives the DHRack. Upon decrypting, and validating the DHRAcknonce, it can create a DH shared secret. The ED can now update it's context with the new keys, and derive a new root key. The next time the ED sends a downlink message, this message should be encrypted under the new context.
- **4**: Lastly, the AS receives the uplink message, registers that the DH_ID has changed, and can then update it's own context with the saved shared secret. The As can then decrypt the uplink message.

4.3.3 Double Ratchet API

We will in this section go through the core functions that make up the double ratcheting protocol library. The library contains two structs, that each are made to represent the mutable state for the ED and the AS, in terms of the Double Ratchet protocol. The implementation for these two structs are largely the same, except for minor parts, that let's them perform the DHRP as described. There are also auxilliary functions that are omitted for the sake of conciseness.

Each of the parties are initialized by supplying a root key, sending and receiving chain key, a Devaddr which is to be used as a connection identifier, and a RNG. The supplying of the RNG allows the caller to supply their own implementation of a RNG, as long as it implements the necessary traits RNGCore, and CryptoRNG, which allows the RNG to be used to generate ECDH keys. In listing 4.5, the full structure representing the ED is shown, the number types are all initialized at 0, and start counting as the protocol moves along. The **mk_skipped** collection is a hashmap, keyed by a tuple of a **dh_id** and a **fcnt** value, allowing retrieval of a stored MK at that point.

```rust
pub struct EDRatchet <Rng: CryptoRng + RngCore>
{
    rk: [u8;32], // root key
    sck: [u8;32], // sending chain key
    rck: [u8;32], // receiving chain key
    pub fcnt_down: u16, // sending message numbering
    pub fcnt_up: u16, // receiving message numbering
    mk_skipped : BTreeMap<(u16, u16), [u8; 32]>, // skipped keys
    // temporary keypair, used if DHRP has been started, but not finalized
    tmp_pkey : Option<PublicKey>,
    tmp_skey : Option<StaticSecret>,
    // Nonces for dhr procedure
    dhr_req_nonce : u16,
    dhr_ack_nonce : u16,
    // DH_ID, indicating the current DHRP session context
    dh_id : u16,
    // devaddr, supplied by the AS during Join procedure
    devaddr : [u8;4],
    // Rng implementation, allowing the caller to supply own rng
    rng : Rng,
}
```

Listing 4.5: ED encryption

After two parties are initialized with corresponding keys, they are now able to encrypt messages for each other, without needing to perform any DHRP steps. For the ED to encrypt uplink messages, it needs to call the publicly available function **ratchet_encrypt_payload**, which is simply a wrapper function that passes the correct Mtype to the **ratchet_encrypt function**, as can be seen in listing 4.6.

```rust
    fn ratchet_encrypt(&mut self, plaintext: &[u8], mtype : i8) -> Vec<u8> {
        let (sck, mk) = kdf_ck(&self.sck);
        self.sck = sck;

        let mut nonce = [0;13];
        self.rng.fill_bytes(&mut nonce);

        let encrypted_data = encrypt(&mk[..16],
                                     &nonce,
                                     plaintext,
                                     &concat(
                                         mtype,
                                         nonce,
                                         self.dh_id,
                                         self.fcnt_up,
                                         self.devaddr));
        let phypayload = PhyPayload::new(mtype,
                                         self.devaddr,
                                         self.fcnt_up,
                                         self.dh_id,
                                         encrypted_data,
                                         nonce);
        self.fcnt_up += 1;
        phypayload.serialize()
    }
```

```
26
27      pub fn ratchet_encrypt_payload(&mut self, plaintext: &[u8])
28      -> Vec<u8> {
29          self.ratchet_encrypt(plaintext,7)
30      }
```

Listing 4.6: ED ratchet encryption

For encryption, a new MK is first derived by advancing the sending chain, in line 2-3, which performs a HKDF_expand on the current SCK. Then the plaintext is encrypted, with a freshly generated nonce, and authenticated on a of all the fields of the PHYPayload concatenated together.

The **encrypt** function in line 8, performs authenticated encryption of the plaintext, using a freshly generated nonce, and passes a concatenation of all the other PHYPayload fields as associated data to be authenticated.

This is all combined to a new PHYPayload, and packed to a parseable stream of bytes.

Note how only the first half of the MK is passed to the encryption. This is because the AES-CCM-16-64-128 takes a 128 bit key, while the HKDF with Sha256 outputs two keys of 256 bits. For this reason the MK is truncated whenever used for encryption or decryption.

Upon receiving a downlink message, the **ratchet_decrypt** function is used. After deserializing the PhyPayload, the ED calls **try_skip_message_keys**, checking whether the incoming **fcnt** and **dh_id** can be found in **mk_skipped**, and if so, attempt to decrypt it with the stored MK.

If the incoming message is not decryptable by any of the stored keys, it will attempt to decrypt it self. On line 8, **skip_message_keys** is called, which checks whether the stored frame counter is larger than the stored one, and skip as many keys as needed, storing them in **mk_skipped**. The receiving chain is then advanced, the down frame counter is incremented, and the message is decrypted with the derived MK.

```
1       fn ratchet_decrypt(&mut self, phypayload: Vec<u8>)
2       -> Result<Vec<u8>, &'static str> {
3           let deserial_phy = deserialize(&phypayload)?;
4
5           match self.try_skipped_message_keys(&deserial_phy)  {
6               Some(out) => Ok(out),
7               None => {
8                   self.skip_message_keys(deserial_phy.fcnt);
9                   let (rck, mk) = kdf_ck(&self.rck);
10                  self.rck = rck;
11                  self.fcnt_down += 1;
12
13                  return decrypt(&mk[..16],
14                      &deserial_phy.nonce,
15                      &deserial_phy.ciphertext,
16                      &concat(deserial_phy.mtype,
17                              deserial_phy.nonce,
18                              deserial_phy.dh_pub_id,
19                              deserial_phy.fcnt,
20                              deserial_phy.devaddr));
21              }
22          }
23      }
```

Listing 4.7: ED ratchet decryption

The ED has the ability to initiate a new DHRP, this is done by calling initiate_ratch(), that simply generates a new keypair, with the provided RNG, saves those keys for later in the tmp_skey and tmp_pkey variable, and then constructs a encrypted DHRReq message:

```
1     pub fn initiate_ratch(&mut self) -> Vec<u8> {
2         let ed_dh_privkey : StaticSecret  = StaticSecret::new(&mut self.rng);
3         let ed_dh_public_key = PublicKey::from(&ed_dh_privkey);
4         self.tmp_pkey = Some(ed_dh_public_key);
5         self.tmp_skey = Some(ed_dh_privkey);
6         self.dhr_req_nonce += 1;
7         let dhr_req = prepare_dhr(ed_dh_public_key.as_bytes(), self.dhr_req_nonce);
8
9         self.ratchet_encrypt(&dhr_req,5);
10        }
```

Listing 4.8: ED DHRP initiation

The constructed PhyPayload is sent to the AS, who receives it with it's own **ratchet** function, as seen in listing 4.9. The Phypayload is deserialized and decrypted. It is then ensured that the DHRReqNonce does not indicate an old DHRReq message. The incoming key is combined with the stored private one to create a shared secret. A DHRAck message is then constructed in the same manner as the DHRReq. It is then encrypted and returned.

```
1     fn ratchet(&mut self, dhr_req_encrypted:Vec<u8>) -> Result<Vec<u8>, &str> {
2         let dhr_serial = self.ratchet_decrypt(dhr_encrypted)?;
3         let dhr_req = unpack_dhr(dhr_serial)?;
4
5         if self.dhr_res_nonce >= dhr_req.nonce{
6             return Err("Received old DHRack");
7         }
8         self.dhr_res_nonce = dhr_req.nonce;
9
10        let mut buf = [0; 32];
11        buf.copy_from_slice(&dhr_req.pk[..32]);
12        let ed_dh_public_key = PublicKey::from(buf);
13
14        let as_dh_privkey : StaticSecret  = StaticSecret::new(&mut self.rng);
15        let as_dh_public_key = PublicKey::from(&as_dh_privkey);
16
17        self.dhr_ack_nonce += 1;
18
19        let dhr_ack = prepare_dhr(&as_dh_public_key.as_bytes(),
20                                  self.dhr_ack_nonce);
21        let dhr_ack_enc = self.ratchet_encrypt(&concat_dhr,6);
22
23        self.tmp_shared_secret = Some(*as_dh_privkey.diffie_hellman(&ed_dh_public_key
      ).as_bytes());
24
25        Ok(dhr_ack_enc)
26    }
```

Listing 4.9: AS DHRP step

The DHRAck is sent as a downlink message from the AS to the ED. The ED then runs it's own ratchet function. The function does the same operations as the ratchet for the AS, in decrypting the PHYPayload, and unpacks the DHRAck message. In line 7 of listing 4.10, the ED is able to create a

shared secret from the secret key it stored earlier, and the public key of the AS. From this shared key, the actual ratcheting step can take place, which happens in line 17 and 18, where the root chain is advanced, producing a new RK, SCK and RCK.

Lastly, it is checked whether more than 500 keys have been skipped, and if so, the *mk_skipped* hashmap will be pruned, removing all keys that are under the oldest DHRP session. At the end, the new keys are set, and frame counters are reset.

```
1     fn ratchet(&mut self, dhr_ack_encrypted:Vec<u8>) -> Result<(),&str> {
2
3         let dhr_ack_serial = self.ratchet_decrypt(dhr_ack_encrypted)?;
4         let dhr_ack =  unpack_dhr(dhr_ack_serial)?;
5         if self.dhr_ack_nonce >= dhr_ack.nonce{
6             return Err("Received old DHRack");
7         }
8         self.dhr_ack_nonce = dhr_ack.nonce;
9
10        let mut buf = [0; 32];
11        buf.copy_from_slice(&dhr_ack.pk[..32]);
12        let as_dh_public_key = PublicKey::from(buf);
13
14
15        let shared_secret = *self.tmp_skey.as_ref().unwrap().diffie_hellman(&
      as_dh_public_key).as_bytes();
16
17        let (rk, sck) = kdf_rk(self.shared_secret, &self.rk);
18        let (rk, rck) = kdf_rk(self.shared_secret,&rk);
19
20
21        if self.mk_skipped.len() > 500 {
22            self.prune_mkskipped();
23        }
24        self.skip_message_keys(10);
25
26        self.dh_id += 1;
27
28        self.rk = rk;
29        self.sck =  sck;
30        self.rck =  rck;
31        self.fcnt_up= 0;
32        self.fcnt_down= 0;
33
34        Ok(())
35    }
```

Listing 4.10: ED DHRP step

Once the ED has finalized the DHRP, the next uplink message will be encrypted under the new context. Once the AS decrypts the uplink message, it will register that the DH_ID has incremented, and finalize the DHRP on it's own side, in the same manner as the ED.

Lastly, for convenience, the ED and AS have each their receive function for receiving Phypayloads. These function evaluate upon the incoming mtype, and handles the messages accordingly. The signatures can be seen in listing 4.11. The ED may respond to a incoming message by returning a decrypted plaintext, if it is a regular downlink message, or a DHRAck, it should return nothing. Therefore the receive function for the ED returns a option of a Vector, allowing it to output a None value.

```rust
// ED receive function signature
 pub fn receive(&mut self,input: Vec<u8>) -> Result<Option<Vec<u8>>,&str>

// AS receive function signature
 pub fn receive(&mut self,input: Vec<u8>) -> Result<(Vec<u8>,bool),&str>
```

Listing 4.11: Receive functions signatures

Whenever the AS receives a message, it should always output something, if the incoming message is a uplink, it should output whatever is decrypted, otherwise, it if receives a DHRReq, then a DHRAck message should be produced. For the AS, the output is returned with a boolean, indicating whether or or not the output should be sent back to the ED.

5 Experiments

In this section of the report, we will go through the two major experiments made in this report. This entails proof-of-concept implementations of LoRaRatchet, running on a ESP32 microcontroller, communicating through TCP, and one implementation on a Raspberry PI, communicating by way of LoRa using a SX1276 external lora module. The main goal of our experiments, is to reason about the relative increase in power consumption that is introduced by having a lower or higher DHRP interval.

The experiments model the case of a ED, continuously sending uplink messages, and a AS receiving and decrypting these messages. Each of the experiments define a constant setting the DHRP interval, such that whenever enough uplink messages have been sent, a DHRP is initiated.

As we were to test the protocol under different DHRP intervals, we needed to decide upon which intervals to choose for our experiments. Inspired by the LoRaRatchet report [8], we opted for six different intervals, forming a logarithmic step growth:

- 1
- 4
- 16
- 64
- 256
- 1024

Choosing these intervals, we are able to achieve a wide range of intervals, without doing unnecessarily many experiments.

The first part of the experiments is the ones that run on the ESP32, these are two-part, one running the protocol over TCP, and another running the protocol locally, with no outwards connection. These experiments showcase the effect of the LoRaRatchet protocol running on a constrained device, specifically looking at the effect on the DHRP interval, and how the power usage of the entire device is affected by it.

Due to the incompatibles with the lora modules, the ESP32 and rust, we had to come up with a different setup for our experiments. As the ESP32 supports wifi, it was possible to set up a experiment utilizing TCP to mediate the protocol instead. Using TCP as a transport however, deviates quite a bit from LoRa. Where LoRa is built to be low-power, enabling the wifi capabilities on the ESP32, consumes roughly 3 times the milliamperes compared to not having wifi enabled, simply by enabling the transmission, and receiving [28]. While running the experiment over TCP involves a lot of overhead compared to LoRa, this still allows us to do measurements that will model the relative cost of a higher or lower DHRP interval.

In order to factor out the cost of running TCP on the ESP32, we therefore also performed a test, running the protocol locally on a single ESP32, acting out both parties, in order to get a understanding of the cost of the "pure" protocol running on the ESP32. with no outwards communication.

The second part of the experiments runs on Raspberry Pi's, and mediate the protocol by way of LoRa. On the Raspberry Pi's the power consumption of the LoRa module itself can be measured, and thus indicating the amount of message transmissions. In order to view the impact of the protocol when deploying in sub-optimal environments, we wanted to compare a test case with 100% success rate, and one with less than that.

Please note that the proof-of-concept implementations in this implementation, do not include the sleep mode of the devices that usually are involved in LoRaWAN. This is mostly a matter of time constraints.

Any meaningful sleeping period of the devices would extend the experiments to have to run for much longer, than what was doable in our time constraints. For a experiment like the ESP32, having a sleeping period would also force a re-initialization of the wifi stack, which takes considerable time. Therefore our experiments are made with thread sleeps in-between uplinks.

5.1 ESP32 experiment setup

In this section, we will go through the setup of the ESP32, and important parts of the implementation of it. The implementation for the TCP-based experiment on the ESP32's can be found at Appendix B.2, under the folder *"/esp32/ed"*, while the AS can be found at *"/esp32/as"*. The ED side of the experiment runs on the ESP32, while the AS side is represented by a computer, acting as a TCP-server also running the protocol.

To start with, all values, such as static keys, suite information, DEVEUI and APPEUI are hardcoded into the code as global static constants. It is possible to read stored values, such a encryption keys, in the ESP32's eFuse. These fields are solely one-time programmable, and would not allow us to perform tests with different keys, and could possibly brick the device. For that reason, we opted to keep them as hardcoded values.

The ESP32 features it's own hardware-based random number generator, that is accessible by a unsafe foreign function interface (ffi) to ESP_IDF. For our purposes, we have used this binding to define a HRNG struct that implement the mandatory *CryptoRNG* and *RngCore* traits. This RNG can then directly be used to derive ephemeral keys, or passed to the Double ratcheting initialization.

The general structure of the TCP-based experiment works by initializing the wifi stack of the ESP32, this involves reading wifi credentials as environment variables, finding the correct network by it's ssid, and then pinging the network to check that connectivity is there.

Once the initialization is done, a TCPstream can be created by connecting to the AS, which is in a listening state. A TCPstream is the object that represents the connection between a local and a remote socket. The ED can then use this connection to write and read data between AS and ED.

Once the connection is established, the EDHOC the ED starts by sending the EDHOC messages, which are packed in a PHYPayload as described in section 4.2.2. The AS produces a Devaddr, and sends it along in the second message. After the Join procedure has been run, and the ratcheting communication can take place.

In listing 5.1, the business logic of the ESP32 is shown. The *stream* object here is the connection the AS, which can be passed around as a mutable reference. In line 5, the output of the Join procedure is retrieved as the initial keys for the ratchet and a Devaddr. In line 9, the ratcheting object for the ED is created, and is passed the HRNG of the ESP32.

It is the loop beginning at line 18 that generates the communication for the duration of the protocol. Before begining the loop, the read timeout is set to 5 seconds, which means that the stream will stop listening if more than 5 seconds go by without anything arriving. This is in line with the set receive windows. Opening a second receive window for TCP does not make sense, as TCP guarantees the successful arrival of all messages

At the beginning of every iteration, a 1 second thread sleep is initiated, in order to simulate a small sleep between each uplink sent. The ED sends a uplink message, and then makes a check to see if the FCntUp has reached the constant limit for whether a DHRP should happen, if this is the case, a ratchet procedure is initiated, and a DHRReq is sent. The ED then listens for a DHRAck in line 29, and uses it to finalize the DHRP in line 34.

```rust
fn handle_connection(stream: &mut TcpStream) -> Result<(), Error> {

    // perform join procedure
    let (ed_sck, ed_rck, ed_rk, devaddr) = match edhoc::join_procedure(stream) {
        Some(join_output) => join_output,
        None => return Ok(()),
    };
    // initialize ratchet
    let mut ratchet = EDRatchet::new(
        ed_rk.try_into().unwrap(),
        ed_rck.try_into().unwrap(),
        ed_sck.try_into().unwrap(),
        devaddr.try_into().unwrap(),
        HRNG,
    );

    stream.set_read_timeout(Some(Duration::from_millis(5000)))?;
    loop {
        thread::sleep(Duration::from_millis(1000));
        let uplink = ratchet.ratchet_encrypt_payload(b"uplink");
        stream.write_all(&uplink)?;
        stream.flush()?;

        if ratchet.fcnt_up >= DHR_CONST {
            let dhr_req = ratchet.initiate_ratch();
            stream.write_all(&dhr_req)?;
            stream.flush()?;
            let mut buf = [0; 64];
            let bytes_read = match stream.read(&mut buf) {
                Ok(bytes) => bytes,
                _ => continue,
            };
            let dhr_ack = &buf[0..bytes_read];
            match ratchet.receive(dhr_ack.to_vec()) {
                Ok(x) => match x {
                    Some(x) => println!("receiving message from server {:?}", x),
                    None => continue,
                },
                Err(s) => {
                    println!("error during receive {}", s);
                    continue;
                }
            };
        } else {
            ... Listen for message, and receive if one comes ...
        }
    }
```

Listing 5.1: ESP32 ED ratcheting loop

TCP is designed to to ensure the successful delivery of messages. Thus it ensures a 100% message TSR. We therefore want to emphasize that the point of running the experiment over TCP is to get a understanding of the relative impact of introducing a higher or lower DHRP interval, under the assumption that all transmitted messages get to their destination.

ESP32 local experiment

As mentioned, another experiment was also run on the ESP32, with no outwards connection, and thus no wifi enabled. The code for this experiment can be found at appendix B.3 under *"ESP32/local-experiment"*. It follows the same structure as the TCP-based experiments. This simply involves setting up both parties, letting them perform the join procedure in a single program, and then encrypting and decrypting messages in a similar manner to the loop in listing 5.1, with a 1 second sleep period where the ED encrypts a message for every iteration, which is decrypted by the AS, and a DHRP happening at a the set interval.

5.1.1 ESP32 experiments measurements

This experiment is run on the ESP32, while a laptop functions as the AS. From this laptop, it was possible for us to surveil that messages were decrypted correctly, and that the DHRP would happen in the correct intervals. The ESP32 is powered through a micro-usb cable connected to a 20.000 mAh battery. Between the battery and the ESP32 is the UM34C USB power meter connected, measuring current, and accumulated mAh drawn to the device.

Figure 5.1 shows the fairly minimal hardware setup required for running our experiments with the ESP32, the UM34C is set to measure for 5 hours, after which it halts. For each of the DHRP intervals that we wanted to measure, the experiment was run for 5 hours, and repeated 3 times to get a representative average.

Figure 5.1: ESP32 measurement setup

The experiment running locally on the ESP32 follows the same physical setup, only without the external server.

5.1.2 UM34C possible error margins

Since we are using a UM34C to measure our power consumption, we should keep in mind the accuracy and power consumption of the device.

The accuracy for the UM34C is 0.8% +3digits, meaning that apart from a 0.8% inaccuracy to be expected, the least significant digit of each ampere can be off by 3. For our measurements, reading mAh, this means that we can expect to have an inaccuracy of 0.8% +3 mAh [9].

5.2 Raspberry Pi experiments setup

Finally, in order to have a working implementation of LoRaRatchet communicating over LoRa, we implemented a AS and a ED on two raspberry Pi's.

This prototype, is built to more closely showcase the working of the LoRaRatchet, including LoRa communication, and a AS that is able to handle several connections. This prototype can be found at appendix B.3, where the ED is found under *"raspberry/ed"* and the AS under *"raspberry/as"*.

Firstly, we set the Lora data rate at the highest legal data rate, which is named as data rate 6, which consists of a bandwidth of 250 kHz, and a spreading factor of 7. This is the highest non-reserved allowed data rate in the European region. It is also the data rate which is used in the LoRaRatchet report as the best possible case [8, p.52] [25]. These values are configurable through our SX1276 driver. We set the transmission power to 14dBm (decibel-milliwatts), which is the legal limit in the European Region [25].

In these experiments, all values such as static keys, length of receive windows, DHRP interval, APPEUI and DEVEUI are read from files on the operating system.

The functioning of the ED runs along the same lines as in the experiment on the ESP32's. However, while conducting early tests, we found that we that transmitting messages rapidly after each other, would result in transmission errors from the module. We found that this issue was mitigated by having a bigger delay between uplinks. After running tests, and probing for issues, we settled at a 10 second sleep, again in order to be able to avoid transmission errors.

The state transition diagram for the ED is displayed in figure 5.2. Every receiving action consist of two receive windows of 5 seconds, with a 2 second sleep between them. Any error encountered during the Join procedure will result in a early termination of the protocol, making the device return to it's starting state.

Once the handshake is successfully finished, a loop is started, running with a 10 second sleep in between each iteration, and encrypting a payload for every iteration. After sending the uplink payload, receive windows are opened, which in our experiments do not result in downlink messages coming from the AS. After listening, the ED will evaluate upon whether or not a DHRP should be initiated.

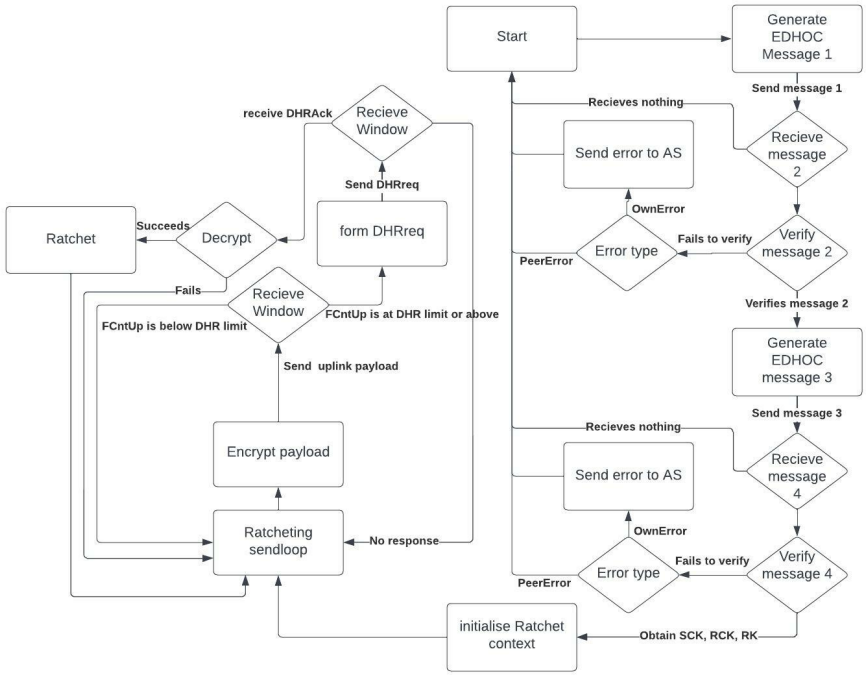

Figure 5.2: ED LoRaRatchet state diagram

The Raspberry Pi AS operates in a dissimilar manner, as it can handle connections to more than one ED at the same time, the state diagram for the AS is displayed in figure 5.3 . The AS is constantly polling for messages which it evaluates them upon their Mtype. The only message which can be received without a appended Devaddr is message 1. After receiving message 1, a DevAddr is generated for the connection. The AS stores the DevAddr, and uses it to correlate incoming messages, with the state of the protocol.

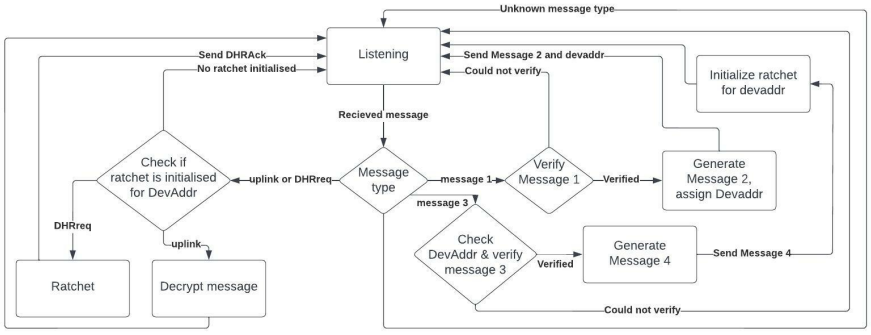

Figure 5.3: AS LoRaRatchet state diagram

5.2.1 Raspberry Pi measurements setup

The wiring needed for this experiment can be seen in table 5.1. When the LoRa device is powered through the Otii Arc power analyzer, GND and 3.3v is connected the to Otii Arc.

And a graphical representation of the pinout can be read in figure 5.4.

Pin Raspberry	Name Raspberry		Name SX1276
06	GND	<->	GND
15	GPIO22	<->	RST
17	3.3	<->	VDD
19	SPI_MOSI	<->	MOSI
21	SPI_MISO	<->	MISO
23	SPI_CLK	<->	SCK
24	SPI_CE0_N	<->	NSS/CS

Table 5.1: Raspberry Pi measurement setup

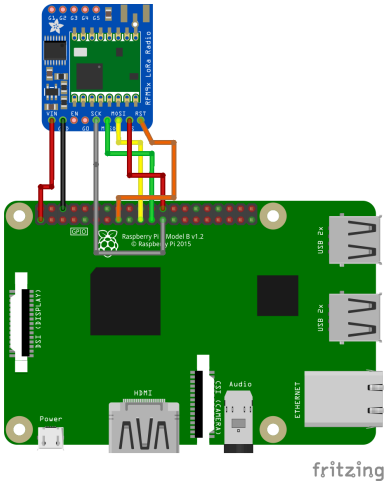

Figure 5.4: Raspberry Pi and SX1276 pinout

As the raspberry Pi's are basically full-fledged linux servers, measuring the power consumption of the devices themselves would be rather futile, in terms of getting readings representing the power consumption of the protocol. What we instead could do was measure the cost of transmissions on the external SX1276 lora module, using the Otii Arc power analyzer.

The physical setup can be seen in figure 5.5, where a Raspberry Pi is powered through a battery, and the Raspberry Pi is connected to our LoRa module, sitting on a small breadboard. In order to measure the power consumption of the LoRa module, the power supply terminals of the Otii Arc is connected to the GND and 3.3v pin on the LoRa module. The Otii Arc is connected to a laptop, running the associated software, which provides the readings.

The AS is not in the picture, but follows the same setup, only without a Otii Arc providing power 5.4.

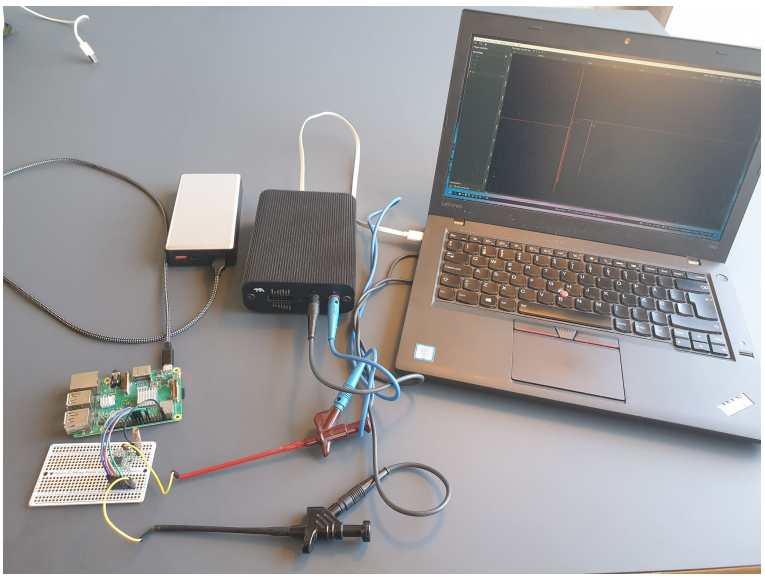

Figure 5.5: Raspberry Pi and SX1276 measurement setup

Each of the Raspberry Pi's have port 22 opened, allowing us to SSH into them and view our experiments running. For example, by setting a counter at each party, counting the amount of sent and received messages. This allowed us to check the transmission success rate, ie. how many messages sent by the ED was received by the AS.

The first round of experiments involved achieving a 100% message transmission success rate . To do this, we simply placed both ED and AS in the same room, roughly 2 meters apart. Unsurprisingly, this results in all messages being relayed successfully.

For the second round of experiments. We needed to reliably introduce a certain amount of message loss. This task proved harder than previously thought. Placing the ED and AS in different rooms with cinderblock walls between them would indeed drop messages, but often at inconsistent rates. Through many trials and errors though, we were able to place the devices in each their room, with two thick walls between them. This proved to be a sweet-spot, and while doing our measurements we could verify that the transmission success rate would stay within the range 75-83%.

Having the Otii Arc connected with the laptop running the associated Otii Arc desktop application, allows us to view the measurements while running. In figure 5.6, the red line represent a ED sending a uplink, signified by the instant surge at 125 mA. After the surge, the reading drops to a stable reading of 12 mA, signifying a first receive window of 5 seconds, it then sleeps for 2, and opening the second 5-second window, with nothing arriving in the windows.

The green line follows the same procedure, but including a DHRP at the end, sending a DHRReq, and quickly receiving a DHRAck, where the SX1276 then goes back into it's sleeping mode.

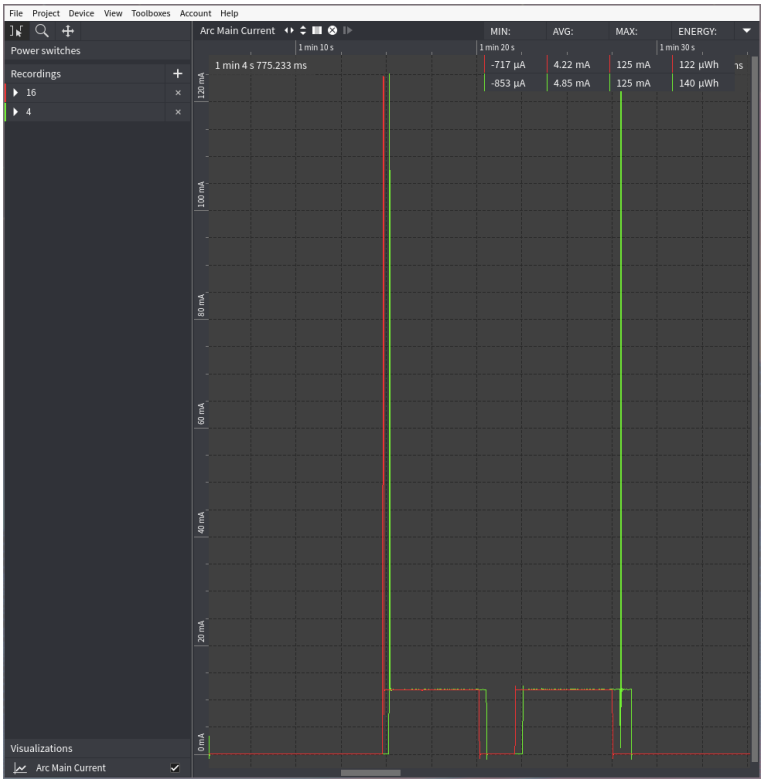

Figure 5.6: Otii Arc reading: uplink and DHRP

As the DHRAck in figure 5.6 is received immediately after sending, the receive window is closed rather quickly. In cases where DHRAck is lost in transmission, the ED would spend far more time listening, spending energy on the open windows with no response. This shows how messages that are lost in transmission takes its toll on the power consumption.

Just as for the other experiments, 5 hour experiments were run for each DHRP interval, and repeated 3 times for each interval. The results are the average of the 3 measurements. After running the experiment for 5 hours, the Otii Arc provides a average mA reading for the period. This average current is multiplied with the amount of hours, to get the accumulated draw by the device in mAh.

5.2.2 Otii Arc possible error margins

The Otii Arc is praised in the embedded community for being highly accurate, despite being easier to use than oscilloscopes. The developers of the device do however provide some data on how much noise can be expected When looking at the datasheet[10], we see that for measurements of a 1 mA, a error margin of 0.11% off the correct value. For lower current measurements, such as 1 uA, a higher disturbance of 5.10% is to be expected [10].

While listening and transmitting, the SX1276 consumes around 10 to 12 mA, and thus for that range, the expected error margin can be said to be quite insignificant. For the lower ranges though, where the SX1276 is in it's sleeping state, it consumes between 0 to 1 uA, then the percentual error margin will be larger. However for a current of 1 uA, this would only be a error of 0.051 uA, which should not throw off our measurements.

6 Results

This section will detail the data produced from running the experiments as described in the previous sections. The main goal is to evaluate upon how a higher or lower DHRP interval affects the consumption. All measurements are repeated 3 times, and the average of these three measurements is averaged for the final result. The repetition of the experiments for 3 measurements, is used to be able to factor out some of the insecurities that are imposed by our measurement devices. Result tables with the raw measured data can be found in appendix A.

ESP32 Results

The first measurements involved running LoRaRatchet on the ESP32, with a 1 second thread sleep in between uplinks. The measurements fall within the range 671.67-629.67 mAh, which is quite high for a constrained device. This is due to the cost of running TCP, with the wifi stack enabled. Regardless of this, the results paint a clear picture, which can be viewed in figure 6.1.

Unsurprisingly, a interval of 1 consumes by far the most power. In the lower intervals of 1-4 and 4-16 the difference is still quite significant, while the curve stagnates as the interval becomes larger than 64.

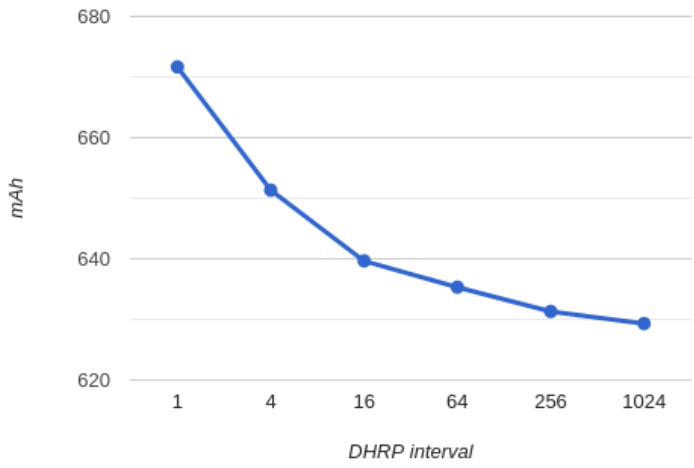

Figure 6.1: ESP32 running LoRaRatchet over TCP (1 second thread sleep)

In table 6.1, the same data is displayed, but showing the relative decrease between intervals, expressed as percentages. It is quite clear that the effect stagnates at a DHRP around 64, and only becomes smaller after that point.

While TCP introduces a considerable amount of overhead. This shows quite clearly how a constrained device running LoRaRatchet is considerably affected by performing regular DHRP's, while the differences between DHRP intervals in the higher ranges are far less consequential.

DHRP interval	**1-4**	**4-16**	**16-64**	**64-256**	**256-1024**	**total decrease**
Percentage mAh decrease	3.12%	1.82%	0.68%	0.63%	0.32%	6.57%

Table 6.1: ESP32 TCP relative decrease mAh power consumption across DHRP intervals (5 hours, 1 second sleep)

The second round of measurements on the ESP32, involves running the local experiment, with no outwards transport involved. The readings are pictured in 6.2. This experiment factors out the cost of having the ESP32 wifi module enabled. This means that the extra power consumption that is induced

by a higher or lower interval, is only the result of performing the extra computations of performing DHRP steps, and skipping 10 keys for every DHRP.

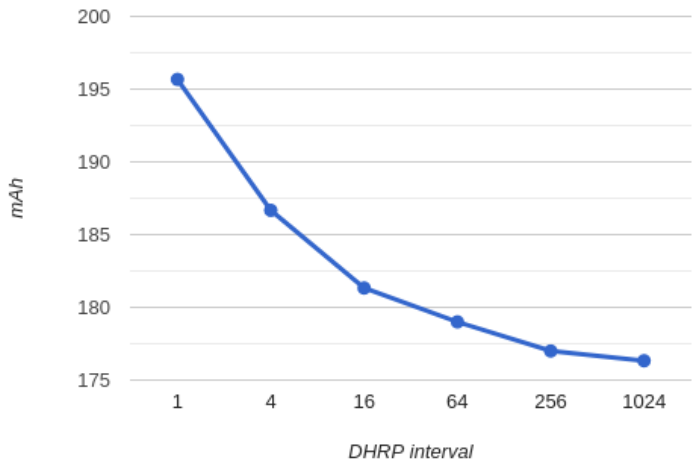

Figure 6.2: ESP32 running LoRaRatchet locally (1 second thread sleep)

Running the protocol solely locally gives us a far more pronounced difference between intervals. This has to do with the wifi being disabled, making the overall power consumption lower. Meaning that both sides of the DHRP is executed on the same device.

The results are fairly intuitive, with stagnation at the higher intervals, compared to running TCP. This is likely due to the fact that we are only viewing the effect of running the cryptographic operations on the ESP32. The results do however still exhibit the general pattern, where the lower intervals have a considerable effect, and the higher intervals imparting a significantly increased usage.

DHRP interval	**1-4**	**4-16**	**16-64**	**64-256**	**256-1024**	**total decrease**
Percentage mAh decrease	4.82%	2.94%	1.30%	1.13%	0.38%	10.57%

Table 6.2: ESP32 local relative decrease mAh power consumption across DHRP intervals (5 hours, 1 second sleep)

Raspberry Pi Results

The second round of experiments was run on Raspberry Pi's communicating via the external SX1276 module. The goal of these experiments is to investigate how differing DHRP intervals affects the power consumption of the lora module itself, and how this may be affected by the TSR being in a range of 75%-83%.

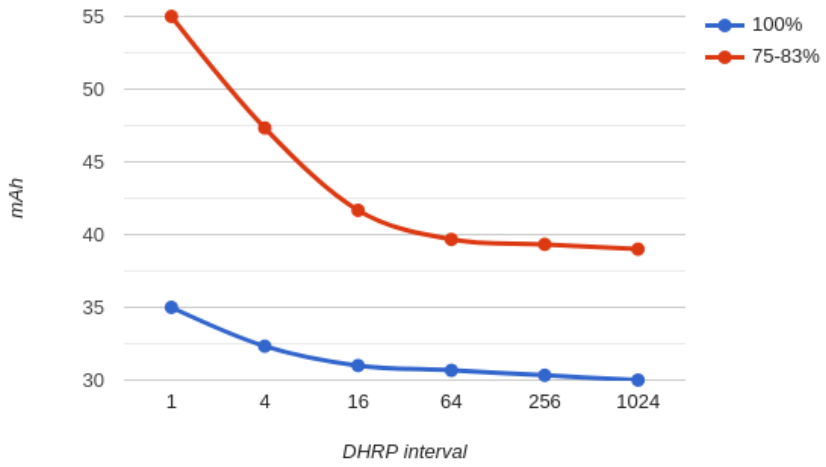

Figure 6.3: Raspberry pi running LoRaRatchet (10 second thread sleep)

Figure 6.3 shows the consumption under the two different TSR's. For the case with all messages being relayed successfully, the data follows a pattern we know, where the curve stagnates around the DHRP of 16, and the percentage decrease are smaller as seen in table 6.3. For the case with only 75 to 83% TSR we see the curve in figure 6.3 stagnate around the DHRP of 64. As seen in the graph, there are big differences when going from a DHRP of 1 to 4, 4 to 16 and a smaller jump going from 16 to 64. From there we see a stagnation, as there is only a small difference in the power readings.

DHRP interval /TSR	**1-4**	**4-16**	**16-64**	**64-256**	**256-1024**	**Total decrease**
100%	7.63%	3.03%	1.08%	1.12%	1.10%	15.22%
75-83%	13.95%	13.58%	5.04%	0.86%	0.85%	34.28%

Table 6.3: Raspberry Pi experiment (relative decrease, 5 hours, 10 second sleep)

From these readings it can be seen, that as soon as there is introduced some loss to the TSR. As seen the loss has the highest impact in the lower DHRP intervals, this is theorised to be due to the higher risk of a DHR request message to be lost, and thus having a receive window opened for a full duration, instead of closing it early which normally happens if it would receive a DHR ack. As a higher DHRP interval is set, the risks of losing the DHR request message is lower, and thus more close in regards to power usage.

6.1 Interpretation of results - Written after group split

When looking at the numbers mentioned above, it can be seen that we start to hit a negligible percentage amount of power savings going from a DHRP interval of 64 to 256. We achieve a power saving from 0.63% to 1.13% in the different test cases.

As noted earlier, the test cases with a good TSR, experienced a small decrease in power usage compared to our testcase with a TSR of 75% to 83%, where we see these big relative decreases in power usage until the DHRP interval are set to 64 and compared to 256 and further. From this its possible to see how a low DHRP interval can manage to decrease the performance, especially with a connection or environment which results in a lower TSR.

When looking at the raw drops in the mAh consumption, there is a more noticeable drop in the local experiment, however its important to note here that the ESP32 both functions as the ED and AS

and thus have a more substantial power draw, compared to the 2 other tests where the devices only functions as the ED. When taking this into consideration, the LoRaRatchet protocol itself does draw a higher amount of power compared to the LoRa communication. We see it stagnate a bit later than the SX1276 LoRa module, it can be seen stagnating more between 256 and 1024.

From these experiments there seems to be a correlation between TSR and a optimal DHRP interval. For the given test cases, this would be 64 for the tested intervals, since we can not always guarantee optimal conditions which results in a good TSR. Even worse TSR than tested here could in theory lead to worse battery performance, since there is a higher risk of loss of the DHRP messages.

These observations means that if the user of LoRaRatchet expects the messages to have a high TSR, a lower DHR interval is recommended as we see the power usage is substantially lower with a higher TSR, and the real power savings from having a higher DHR interval are coming from the scenarios where we see a lower TSR. The DHR interval should also be choosen depending on the value of the data that are being sent and how important the savings of a few mAh are. Depending on these factors a DHR interval at either 64 or 256 seems to be the most optimal interval.

7 Conclusion - Written after group split

In conclusion, this thesis has shown that it is feasible to implement and run LoRaRatchet on constrained devices. As seen in chapter 4 the LoRaRatchet protocol was implemented. As explained in chapter 5 and 6 its seen running on a ESP32 as an ED, and communicating with a AS, sending messages back and forth. The communication protocol was abstracted away from LoRa and TCP were used instead, due to issues described. This showed that the protocol itself was possible to run on a constrained device, while being lightweight enough so there was no noticeable delay on ratchets or EDHOC procedures, however this was only a "look at the DHRP" and not on a scientific ground.

As seen in chapter 6, the effect of our DHRP interval can have an impact on battery usage, both from the compute side of things, and as a side effect with receive windows and the LoRa modules used for communication. As talked about in chapter 6.1, there seems to be a correlation between the DHRP interval, the battery consumption and the TSR. I expect the most optimal DHRP interval to be around 64, but still dependent on the factors listed in section 6.1. This is based on limiting the amount of messages being sent on the same racthet chain, while still having a power saving measure. By choosing this interval there is the lower risk of not losing a DHRP message, and thus limit the power usage while still regularly ratcheting.

This thesis could be expanded upon in several ways, in future works.

- Benchmarks on the libraries themselves, to gauge the running times of the primitives that make up the protocol
- Combine EDHOC and double ratchet library to a single library, which makes the LoRaRatchet implementation easier for developers.
- An in depth formal verification of the LoRaRatchet protocol

These points could be interesting to further work upon, on the given ground that right now this implementation is a proof of concept. Where the focus is, is it even able to run on constrained devices and at what cost of the battery life. A benchmark of the protocol could be interesting to see if it was also viable on lower end constrained devices compared to what we used. Implementing a version which is less proof of concept and more friendly to developers would make the protocol more viable for actual use and thus actually might gain some ground.

References

[1] Carsten Bormann. *Concise Binary Object Representation (CBOR) Sequences*. RFC 8742. Feb. 2020. DOI: 10.17487/RFC8742. URL: https://www.rfc-editor.org/info/rfc8742.

[2] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Oct. 2013. DOI: 10.17487/RFC7049. URL: https://www.rfc-editor.org/info/rfc7049.

[3] David Martin Carl. *sx127x_lora*. https://github.com/DavidCarl/sx127x_lora. 2022.

[4] Esteban Martinez Fayo Cesar Cerrudo and Matias Sequeira. 'LoRaWAN Networks Susceptible to Hacking: Common Cyber Security Problems, How to Detect and Prevent Them'. In: (2020), p. 10. URL: https://act-on.ioactive.com/acton/attachment/34793/f-87b45f5f-f181-44fc-82a8-8e53c501dc4e/1/-/-/-/-/LoRaWAN%5C%20Networks%5C%20Susceptible%5C%20to%5C%20Hacking.pdf.

[5] *cross*. https://github.com/cross-rs/cross. 2021.

[6] Matin Disch. *Lightweight Application Layer Protection for Embedded Devices with a Safe Programming Language*. https://github.com/martindisch/oscore. 2019.

[7] Mohamed Eldefrawy et al. 'Formal security analysis of LoRaWAN'. In: *Computer Networks* 148 (2019), pp. 328–339. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2018.11.017. URL: https://www.sciencedirect.com/science/article/pii/S1389128618306145.

[8] Astrid Ellermann-Aarslev and Laura Kromann-Larsen. 'Ratcheting in LoRaWAN: Investigating the viability and feasibility of implementing the Double Ratchet algorithm in constrained networks'. In: (2021).

[9] Ltd HangZhou RuiDeng Technologies Co. *"UM34C Datasheet"*. 2018. URL: https://jucetize.weebly.com/uploads/3/7/2/0/37200949/um34_c__usb_tester_meter__instruction___android_app_instruction-_2_in_1_-__2018.6.13_.pdf.

[10] *How accurate is your low current measurement?* URL: https://www.qoitech.com/blog/how-accurate-is-your-current-measurement/.

[11] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019. URL: https://doc.rust-lang.org/book/.

[12] Dr. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869.

[13] Hugo Krawczyk. 'SIGMA: the 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols'. In: (2003).

[14] *Nabto)*. https://www.nabto.com/guide-to-iot-esp-32/. Accessed: 2022-04-18.

[15] Karl Norrman, Vaishnavi Sundararajan and Alessandro Bruni. 'Formal Analysis of EDHOC Key Establishment for Constrained IoT Devices'. In: *CoRR* abs/2007.11427 (2020). arXiv: 2007.11427. URL: https://arxiv.org/abs/2007.11427.

[16] *Otii Arc Documentation*. https://www.qoitech.com/docs/user-manual/otii. 2022.

[17] Trevor Perrin and Moxie Marlinspike. *Signal*. https://signal.org/docs/. Accessed: 2012-03-15.

[18] *Raspberry Pi 3 Model B+*. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. 2020.

[19] *rust-build)*. https://github.com/esp-rs/rust-build#rust-on-xtensa-installation-for-linux. Accessed: 2022-04-18.

[20] Jim Schaad. *CBOR Object Signing and Encryption (COSE)*. RFC 8152. July 2017. DOI: 10.17487/RFC8152. URL: https://www.rfc-editor.org/info/rfc8152.

[21] Göran Selander, John Preuß Mattsson and Francesca Palombini. *Ephemeral Diffie-Hellman Over COSE (EDHOC)*. Internet-Draft draft-selander-ace-cose-ecdhe-14. Work in Progress. Internet Engineering Task Force, Oct. 2019. 80 pp. URL: https://datatracker.ietf.org/doc/html/draft-selander-ace-cose-ecdhe-14.

[22] Göran Selander, John Preuß Mattsson and Francesca Palombini. *Ephemeral Diffie-Hellman Over COSE (EDHOC)*. Internet-Draft draft-ietf-lake-edhoc-12. Work in Progress. Internet Engineering Task Force, Oct. 2021. 80 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-lake-edhoc-12.

[23] Semtech. 'LoRa® and LoRaWAN®: A Technical Overview'. In: *Semtech Corporation* (2019). URL: https://lora-developers.semtech.com/uploads/documents/files/LoRa_and_LoRaWAN-A_Tech_Overview-Downloadable.pdf.

[24] Semtech. 'LoRaWAN 1.1 Specification'. In: *LoRa Alliance* (2017). URL: https://lora-alliance.org/wp-content/uploads/2020/11/lorawantm_specification_-v1.1.pdf.

[25] Semtech. 'RP002-1.0.1 LoRaWAN® Regional Parameters'. In: *LoRa Alliance* (2020). URL: https://lora-alliance.org/resource_hub/rp2-102-lorawan-regional-parameters/.

[26] Semtech. *SX1276/77/78/79 Datasheet*. 2015. URL: https://cdn-shop.adafruit.com/product-files/3179/sx1276_77_78_79.pdf.

[27] *State of the IoT 2020: 12 billion IoT connections, surpassing non-IoT for the first time*. https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/. Accessed: 2022-10-03.

[28] Espressif Systems. *"ESP32 Series: Data sheet"*. Version 3.9. 2022.

[29] *The things Network: Duty cycle*. https://www.thethingsnetwork.org/docs/lorawan/duty-cycle/. Accessed: 2022-04-14.

[30] *The things Network: spreading factors*. https://www.thethingsnetwork.org/docs/lorawan/spreading-factors/. Accessed: 2022-03-29.

[31] Charles Wade. *sx127x_lora*. https://github.com/mr-glt/sx127x*lora*. 2018.

[32] Xueying Yang et al. 'Security Vulnerabilities in LoRaWAN: A Technical Overview'. In: *Conference on Internet-of-Things Design and Implementation* (2018), pp. 3–12. DOI: DOI10.1109/IoTDI.2018.00022.

[33] Ilsun You et al. 'An Enhanced LoRaWAN Security Protocol for Privacy Preservation in IoT with a Case Study on a Smart Factory-Enabled Parking System'. In: June 2018. DOI: 10.3390/s18061888.

[34] Ilsun You et al. 'An Enhanced LoRaWAN Security Protocol for Privacy Preservation in IoT with a Case Study on a Smart Factory-Enabled Parking System'. In: *Sensors (Basel, Switzerland)* 18 (2018).

A Result tables

This appendix section includes tables showing the actual measured power consumptions from the experiments in the report, measured by the mAh, over the lifetime of the experiments run. All individual experiments are run three times, and the final result is the average of those three runs.

DHRP interval /Experiment run	**1**	**4**	**16**	**64**	**256**	**1024**
1st	666	648	638	631	627	633
2nd	675	647	636	639	638	628
3rd	674	659	645	636	629	627
Average	671.67	651.33	639.67	635.33	631.33	629.33

Table A.1: ESP32 TCP experiment (results in mAh usage and over a 5 hour period)

DHRP interval /Experiment run	**1**	**4**	**16**	**64**	**256**	**1024**
1st	194	183	179	178	177	178
2nd	196	190	185	181	179	176
3rd	197	187	180	178	175	175
Average	195.67	186.67	181.33	179.00	177.00	176.33

Table A.2: ESP32 local experiment (results in mAh usage and over a 5 hour period)

DHRP interval /Experiment run	**1**	**4**	**16**	**64**	**256**	**1024**
1st	33	32	30	29	31	30
2nd	36	35	30	31	28	27
3rd	36	30	33	32	32	33
Average	35.00	32.33	31.00	30.67	30.33	30.00

Table A.3: Raspberry Pi experiment data for 100% TSR in mAh

DHRP interval /Experiment run	**1**	**4**	**16**	**64**	**256**	**1024**
1st	49	40	49	44	45	43
2nd	64	53	39	33	35	33
3rd	52	49	37	42	38	41
Average	55.00	47.33	41.67	39.67	39.33	39.00

Table A.4: Raspberry Pi experiment data for 75-83% TSR in mAh

B Repositories

1. **EDHOC implementation**: `https://github.com/DavidCarl/edhoc`
2. **Double ratchet implementation**: `https://github.com/S3j5b0/Double_ratchet`
3. **Repository containing experiments**: `https://github.com/DavidCarl/loraRatchetThesis`