École Polytechnique Fédérale de Lausanne

Security mechanisms on Bluetooth Low Energy

by Bradley Mathez

Semester Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Prof. Daniele Antonioli
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 11, 2021

Acknowledgments

I would like to express my deep gratitude to my semester project's supervisor, Professor Daniele Antonioli, for his expertise, very appreciated comments and enormous support during the whole research, without whom none of this would have been possible in such a short and complicated period. I have learned a lot since the beginning of this project, especially through Daniele Antonioli's guidance but also with the advices and help coming from the other team members, Alessandro Bianchi, Emiljano Gjiriti, Francesco Berla and Jean-Michel Crepel. I am further grateful to HexHive at EPFL and Prof. Dr. sc. ETH Mathias Payer for making this semester project realizable. The subject of this report was challenging and extremely interesting, thank you.

Lausanne, June 11, 2021 Bradley Mathez

Abstract

The Bluetooth standard has kept security flaws through multiple versions to stay retro-compatible. This is dangerous since every standard-compliant Bluetooth devices are then currently vulnerable to these weaknesses. From another point of view, there now exists modern solutions employed to exchange confidential data between two parties such as the Double Ratchet algorithm. Presently, this method is integrated to many different messaging applications, for instance WhatsApp or Signal. In our project we started the assessment of the possibility to introduce this algorithm in the Bluetooth communication protocol through the development of a *C* implementation of the Double Ratchet algorithm.

Contents

Chapter 1

Introduction

This project aims to summarize the results achieved throughout six months of research and development on how to improve Bluetooth security using the Double Ratchet algorithm. The reader can find the code related to this report at that URL: *https://github.com/Dreyvor/HEXHIVE-semester_project*

The first section discusses the motivations and the background of this research. We will present a fast overview of the Bluetooth technologies and employ this knowledge to justify the ambitions behind this operation. We will also explain the Double Ratchet algorithm, to make the reader familiar with it. These points are crucial as this project exploits this algorithm as a basis for development.

The second paragraph serves the purpose of spelling out the design and some choices made before and during the development phase.

The third segment addresses the implementation, and the challenges we encountered.

Finally, the last section exposes the conclusion with a short summary of what would have been discussed and proposes paths for future works.

Chapter 2

Motivation and Background

In order to understand the motivations that drove this project, the reader must know the technologies offered by Bluetooth and their vulnerabilities. Bluetooth is a short-range wireless technology widely used by many products ranging from mobile devices to industrial systems and personal computers. Introduced in 1989, it has been developed to replace cables.

2.1 Bluetooth and vulnerabilities

Bluetooth provides two protocols:

1. Bluetooth Classic (BT): This is the first Bluetooth protocol. It offers high throughput persistent connections (e.g., used by speakers, headphones, keyboards, printers or while transferring data between cellular phones)
2. Bluetooth Low Energy (BLE): It appeared later and is designed for applications that need to periodically transmit data (e.g., used by GPS trackers, monitoring sensors, indoor navigation services). Thus, it consumes a lot less power than BT.

These technologies have to negotiate and exchange a pairing key and a session establishment protocol. Both contain vulnerabilities. For example, during the 28th USENIX Security Symposium in 2019, Antonioli et al. presented their research on the KNOB attack [3]. The next year, they submitted the BIAS attack in the IEEE Symposium on Security and Privacy [1]. Each is standard-compliant making them particularly critical:

1. KNOB attack: can be conducted on BT and BLE [2] [3]. It decreases the entropy of the key established between the devices to something that a computer can brute force these days.

2. BIAS attack: can be performed on BT [1]. It targets the authentication phase of secure connection establishment and allows an attacker to impersonate any systems.

With the attacks, we cannot assure the integrity or the confidentiality of the data exchanged through BT or BLE. This is worrying on such a widespread technology. As we will see in the next section (section 2.2), the Double Ratchet algorithm may solve this issue by making Bluetooth secure-by-design and comparable to the state of the art communication protocols on the security aspect.

2.2 Double Ratchet algorithm

2.2.1 Overview

The Double Ratchet algorithm (DR) is a key management algorithm and has been created in 2013. It's used to exchange data between two parties. Multiple messaging applications are utilizing it such as Signal, WhatsApp, Skype, Viber and many others. The DR offers an end-to-end encryption with basic cryptographic properties such as confidentiality, integrity and authenticity. This is a more complete process providing the following characteristics as well:

1. Forward security: The previously exchanged messages are undecipherable even for an attacker that learns a long-term secret used in the encryption.
2. Break-in recovery: The future communications remain confidential even if an adversary determines a secret key employed to derive the future encryption keys. This property ensures that the algorithm can self-heal in that case.

The first property preserves the past messages while the second assures the protection of the future ones. We will now dig deeper in the details of this algorithm as it is necessary for the reader to be familiar with these concepts to fully understand how our implementation is functioning. Our work has been based on the official documentation of the Double Ratchet algorithm provided by Signal [13].

2.2.2 KDF Chains - ratchets

The idea of key derivation function (*KDF*) is a core concept in the Double Ratchet algorithm. This is a cryptographic function that takes a secret, a random KDF key, some input data and that returns a fixed length output. If we assume the input key unknown, then the output data is indistinguishable from random (i.e., it respects the forward security).

To form a KDF chain, we split the output of the KDF in two distinct parts. The first one will be used as the input of the next KDF and the second is the output key and can be employed as an encryption key, for example. In practice, this is a bit more complex.

In fact, a *ratchet* can be defined as a KDF chain. It provides the following properties [13]:

1. Resilience: The output keys appear random to an adversary without knowledge of the KDF keys. This is true even if the attacker can control the KDF inputs.
2. Forward security: Output keys from the past seem random to an adversary who learns the KDF key at some point in time.
3. Break-in recovery: Future output keys appear random to an adversary who determines the KDF key at some point in time, provided that future inputs have added sufficient entropy.

The reader may now have a better understanding of the properties offered by the Double Ratchet algorithm.

2.2.3 Use of symmetric-key ratchets

In a Double Ratchet session, the two parties (e.g., Alice and Bob) use four chains: a *root chain*, a *sending chain* and a *receiving chain* where Alice's receiving ratchet matches Bob's sending ratchet and vice versa. The last chain is the *Diffie-Hellmann chain*. We will explain it later (subsection 2.2.4) on as it brings the last property we will need, after having set the core of the protocol. This section will focus on the first three chains as the sending and receiving ratchets form the *symmetric-key ratchet*. They ensure that each exchanged message is encrypted with a unique key that can be deleted after being used.

Both sending and receiving chains perform in a similar way. They take input data and a chain key (ck_0) as input key. Then, the KDF produces the next chain key (ck_1) and a message key (mk_1). Afterwards, the message key mk_1 feeds an HKDF, which outputs data that is split in three pieces to obtain ek_1 (encryption key), ak_1 (authentication key) and iv_1 (initialization vector). As a consequence, we are now able to build an *authenticated encryption with associated data* (*AEAD*) but only for one message (more details are available in the first appendix with the name).

In the case of the sending and receiving chains, they take a constant as input data. Therefore, we may notice that they do not provide break-in recovery. The algorithm makes use of the Diffie-Hellmann algorithm to deal with this issue (subsection 2.2.4).

Nevertheless, message keys are not used to derive any keys. Thus, they may safely be stored without decreasing the security of other messages keys. We will remember this property later on to deal with out-of-order messages (subsection 2.2.5).

In order to initialize the sending and receiving chains with their respective first derivation key, we use the root chain. It takes messages keys output by the Diffie-Hellmann chain as input data. Its very first input key (i.e., the one initializing the ratchet) must be a shared secret between the two parties. It must be secret and authenticated in order to prevent *man in the middle* attacks at the session establishment. As it is a well-known process and it is out of the scope of this project, we will assume that the shared secret used to initialize the protocol respects the properties we discussed and let the reader implement this part.

2.2.4 Diffie-Hellmann ratchet

To prevent the lack of break-in recovery property and to self heal in case one of the ck_i is discovered, the algorithm uses the Diffie-Hellmann ratchet. This step is based on the computation of a Diffie-Hellmann shared secret between the two parties. In the standard version of the Double Ratchet algorithm, fresh Diffie-Hellmann key pairs are generated each time non-consecutive messages are exchanged (e.g., Alice sends multiple messages to Bob with the same DH key DH_1, then Bob answers through many messages utilizing a new key DH_2. Afterwards, Alice contacts Bob with a last message with a fresh key DH_3). The public Diffie-Hellmann key employed to produce the root key chain is communicated along the message in the header. It does not cause any problem since the delivered key is the public component of the key pair.

Earlier, we already said that the root ratchet took the output of the Diffie-Hellmann chain. Thus, we are now able to understand the big picture of the whole scheme of the Double Ratchet algorithm combining the symmetric-key ratchets with the Diffie-Hellmann ratchet to obtain a system highly secured respecting the properties discussed before (subsection 2.2.1).

The Diffie-Hellmann ratchet generates an output dh_1 used by the root ratchet which is initialized with an authenticated shared secret rk_0 before the initialization of the protocol. The Diffie-Hellmann ratchet makes a step and the root chain produces ck_1 which can be used to initialize the sending or the receiving chain and generate a key mk_1. This output will feed a HKDF to produce three separated keys for an AEAD scheme. Then, the other chain will be set up when a new ck_i will be generated by the Diffie-Hellmann ratchet upon the reception of a message with a header containing a new Diffie-Hellmann public key, and so on.

This process denies an adversary that learned ck_i and even rk_i (where $i > 0$). Since chains keys are often replaced, acquiring some knowledge about a chain key only permits to decrypt messages until a Diffie-Hellmann ratchet step is performed.

2.2.5 Out-of-order messages

Due to the forward security property and the fact that we delete the message keys when they have been used after encryption or decryption, the delayed or lost messages are a problem.

However, we want our system to be resilient against out-of-order messages.

Previously (subsection 2.2.3), we mentioned that message keys can safely be stored without impacting the security of other message keys. This property is essential to make sure that no message is lost even if some arrive being delayed. We send the following three fields within the header of each message:

1. N: represents the message number in the sending chain. It allows to know which message key to use if we never reinitialize the symmetric ratchet through Diffie-Hellmann ratchet's steps. However, we do reset these chains. This is why we need the next field PN
2. PN: represents the length of the previous sending chain. Thus, the receiver knows how much message keys it has to generate before doing a step forward in the Diffie-Hellmann ratchet and resetting the symmetric ratchet.
3. Diffie-Hellmann public key: is the current Diffie-Hellmann public key of the sender. We already discussed this in the previous section (subsection 2.2.4).

In the classic Double Ratchet algorithm, the header is not encrypted but its integrity is protected. In the present settings, when a late message arrives, the message keys are computed and stored until the Double Ratchet reaches the position described by the header's fields.

Chapter 3

Design

The initial goal of this project was to implement the Double Ratchet algorithm at different levels in the BLE protocol and provides metrics about these implementations to study and compare their feasibility and scalability. There are four possibilities that we called "*The Four Blesses*":

1. Bless 1: Ignore the actual Bluetooth's security and implement our solution as the only safety to exchange messages at the application layer while using an authenticated key exchange (AKE) protocol to initialize the Double Ratchet algorithm.
2. Bless 2: Let the pairing being performed and employ the pairing key as AKE to implement our solution in the link-layer.
3. Bless 3: Accept the pairing and the session being established, then utilize the session key as AKE to implement our solution in the link-layer.
4. Bless 4: Add the Double Ratchet algorithm on top of the Bluetooth security in the application layer.

However, we have to implement the Double Ratchet algorithm before interfacing it with the current security. The development board Nordic nRF52840-DK [14] has been chosen as the target platform, because we can flash it with our own Bluetooth software and modify its kernel as we want. In order to avoid developing an operation system and an actual standard-compliant implementation of Bluetooth, we started from the open source *mynewt* systems (newt [6], core [4], NimBLE [7], mcumgr [5]) developed by Apache.

The Double Ratchet's algorithm documentation [13] mentions some possible cryptographic functions that may be employed in our implementation. We are not cryptanalysts and the implementation of our own basic cryptographic functions is outside the scope of this project. Thus, we reused the implementation already present in the *mynewt-core* repository. The provided libraries are *mbedtls* [8] and *tinycrypt* [11]. We have chosen the following functions:

- Curve25519 from mbedtls as elliptic curve for the Diffie-Hellmann protocol.
- SHA-256 from mbedtls as a hash function to implement HKDF.
- SHA-256 from tinycrypt as a hash function to implement HMAC, because this function was already ready to use in tinycrypt but not in mbedtls.
- AES-256 in CBC mode with PKCS#7 padding from mbedtls for the encryption part.

Ultimately, we completed the implementation of the Double Ratchet algorithm (without header encryption). We tested our project through a lot of tests, and it behaves correctly according to Signal's specifications [13]. We also wrote a proper documentation for each function to speed up the workflow of the next developer continuing this research. This documentation is available on the project's repository along with the code.

Unfortunately, we started the implementation of the Bless 4 but we did not have enough time to entirely debug it in order to evaluate it or to present a proof of concept of our project.

Chapter 4

Implementation

Our research offers a tested and documented library developed in *C* of the Double Ratchet algorithm without header encryption. The reader should be able to easily understand how to use it as code base for future projects. Globally, we followed the structure proposed in the Double Ratchet algorithm documentation [13]. However, during the development, we have taken some decisions and we would like to present the reader a part of them.

Firstly, we want to discuss the emulation of a dictionary with a tuple as key in *C*. A such object is needed to keep track of the out-of-order message keys. In this situation the program must store a message key while indexing it with the attached public key and its message number. The main issue is that there is no standard dictionary in *C* and anything is close to a dictionary with a tuple as key. We chose to build a linked list where each node will contain a pointer to the head of a second linked list (a schema is available in the appendix with the name of *MSKIPPED*). The first level of linked list contains the information about the public key where the second level contains the message number and the key associated. This detail of implementation is useful since the reader may continue the development if this research.

Secondly, we wanted our code to be as flexible as feasible to let the future developers a large field of possibilities. Thus, we employed pre-compilation instructions such as `#ifdef/#endif` blocks to easily activate or not some code parts from a config file (called *config.h*).

Thirdly, we also had to alter the mynewt-core and mynewt-nimble code to make our project work. As these modifications are not game changers, we won't describe these in detail in this report. However the reader will find patches ready to apply along with the code. We can mention the increase of the stack or some fixes applied that were not present in mynewt-core or mynewt-nimble releases.

In addition, we started the implementation of the Bless 4 in order to obtain a proof of concept, but as discussed before, we did not have enough time to debug it. During this part, we assumed the initial shared root key secretly shared and hard-coded it in the beginning of the

implementation of Bless 4. Sadly, we only have a fully exploitable API for the Double Ratchet algorithm which has been tested and documented.

Finally, the idea was to make this project available as an easily understandable code base for future research, we wrote a lot of tests and documented every single function and header file following the standard notation used by *Doxygen*[10]. The documentation is available along this project under HTML format.

Chapter 5

Conclusion

This project aimed at providing a *C* implementation of the Double Ratchet algorithm which can later be used for future research and to determine the feasibility and the cost of integrating this algorithm in the Bluetooth standard. The goal was initially to propose an implementation, to evaluate it through multiple metrics and to build proofs of concept at application and link layers with this API. Unfortunately, we only were able to achieve the development and testing of the API without managing the measurements and proofs of concept parts. Remembering indeed that the Bluetooth devices are still vulnerable to standard-compliant attacks [1], [3], [12], this research contributes to the possible beginning of a more secure Bluetooth standard. This work has already begun insidiously [9].

As we already discussed, this project must be continued with the implementation and evaluation of the proofs of concept at the different layers. This is a key step before integrating the Double Ratchet algorithm in the Bluetooth standard. Other important features are the header encryption as well as the authenticated key exchange to initiate the algorithm.

To conclude, there is still a lot of work to be done before modifying the standard and these types of decisions need to be well thought out. However, if the Double Ratchet algorithm becomes incorporated in the new standards of Bluetooth and Bluetooth Low Energy, then it will protect them against many types of attacks such as the BIAS attack [1] or the KNOB attack [3].

Bibliography

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. “BIAS: Bluetooth Impersonation AttackS”. In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. May 2020.

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. *Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy*. https://doi.org/10.1145/3394497. New York, NY, USA, June 2020.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. “The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR”. In: *Proceedings of the USENIX Security Symposium (USENIX Security)*. Aug. 2019.

[4] Apache. *MyNewt-core github repository*. https://github.com/apache/mynewt-core/. Feb. 2021.

[5] Apache. *MyNewt-mcumgr github repository*. https://github.com/apache/mynewt-mcumgr/. Feb. 2021.

[6] Apache. *MyNewt-newt github repository*. https://github.com/apache/mynewt-newt/. Feb. 2021.

[7] Apache. *MyNewt-nimble github repository*. https://github.com/apache/mynewt-nimble/. Feb. 2021.

[8] ARMmbed. *Mbedtls github repository*. https://github.com/ARMmbed/mbedtls. Mar. 2021.

[9] Alessandro Bianchi. “Double Ratchet for Bluetooth Security”. 2021.

[10] Doxygen. *Doxygen documentation*. https://www.doxygen.nl/manual/index.html. May 2021.

[11] Intel. *Tinycrypt github repository*. https://github.com/intel/tinycrypt. Mar. 2021.

[12] Madison Oliver. *Devices supporting Bluetooth Core and Mesh Specifications are vulnerable to impersonation attacks and AuthValue disclosure*. https://kb.cert.org/vuls/id/799380. May 2021.

[13] Trevor Perrin and Moxie Marlinspike. *Signal » Specifications » The Double Ratchet Algorithm*. https://signal.org/docs/specifications/doubleratchet/. Feb. 2021.

[14] Nordic Semiconductor. *Nordic Semiconductor Infocenter*. https://infocenter.nordicsemi.com/topic/struct_nrf52/struct/nrf52840.html. Feb. 2021.

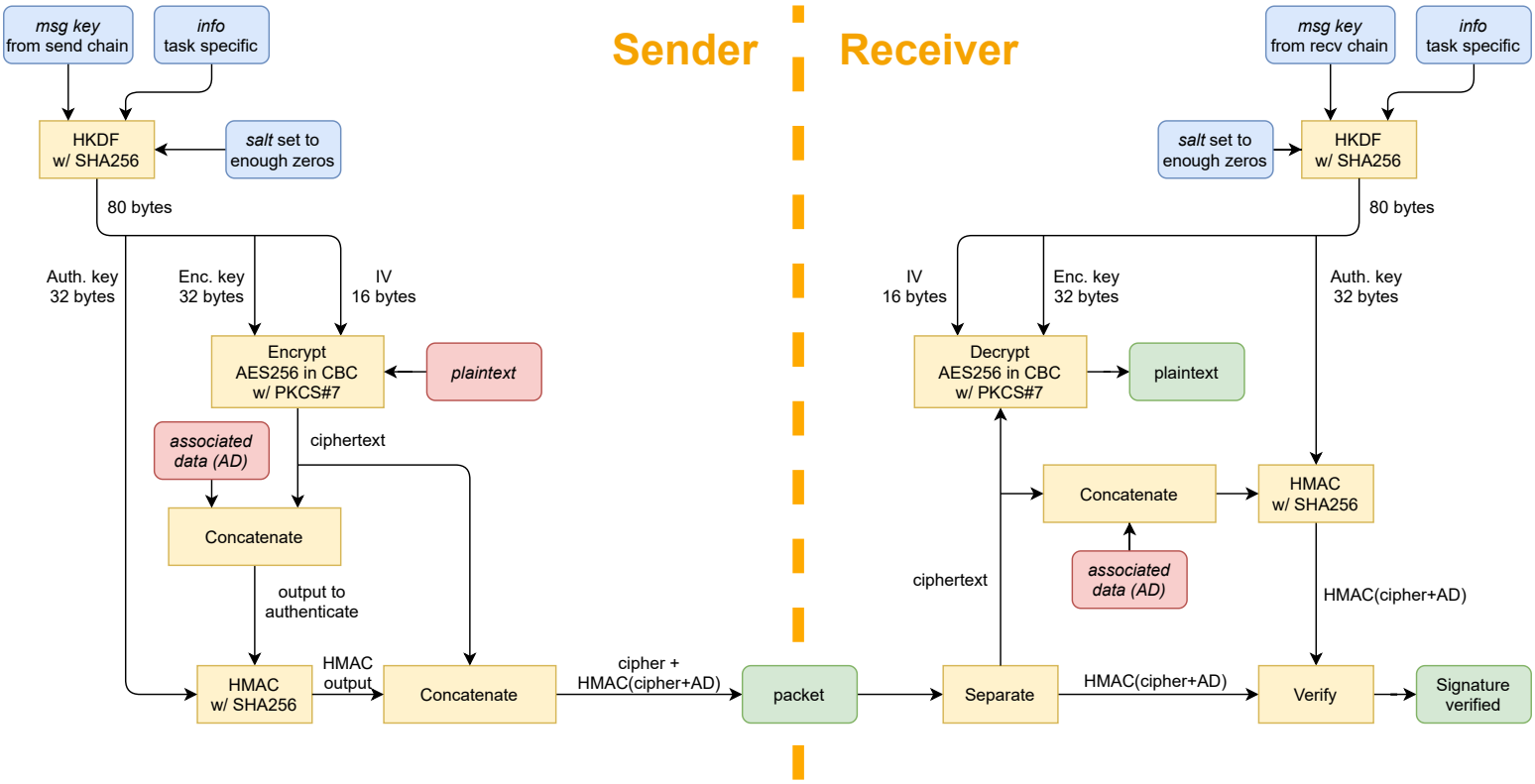
Appendix A

Additional diagrams

The reader will here find more information about the appendix elements in order of their apparition.

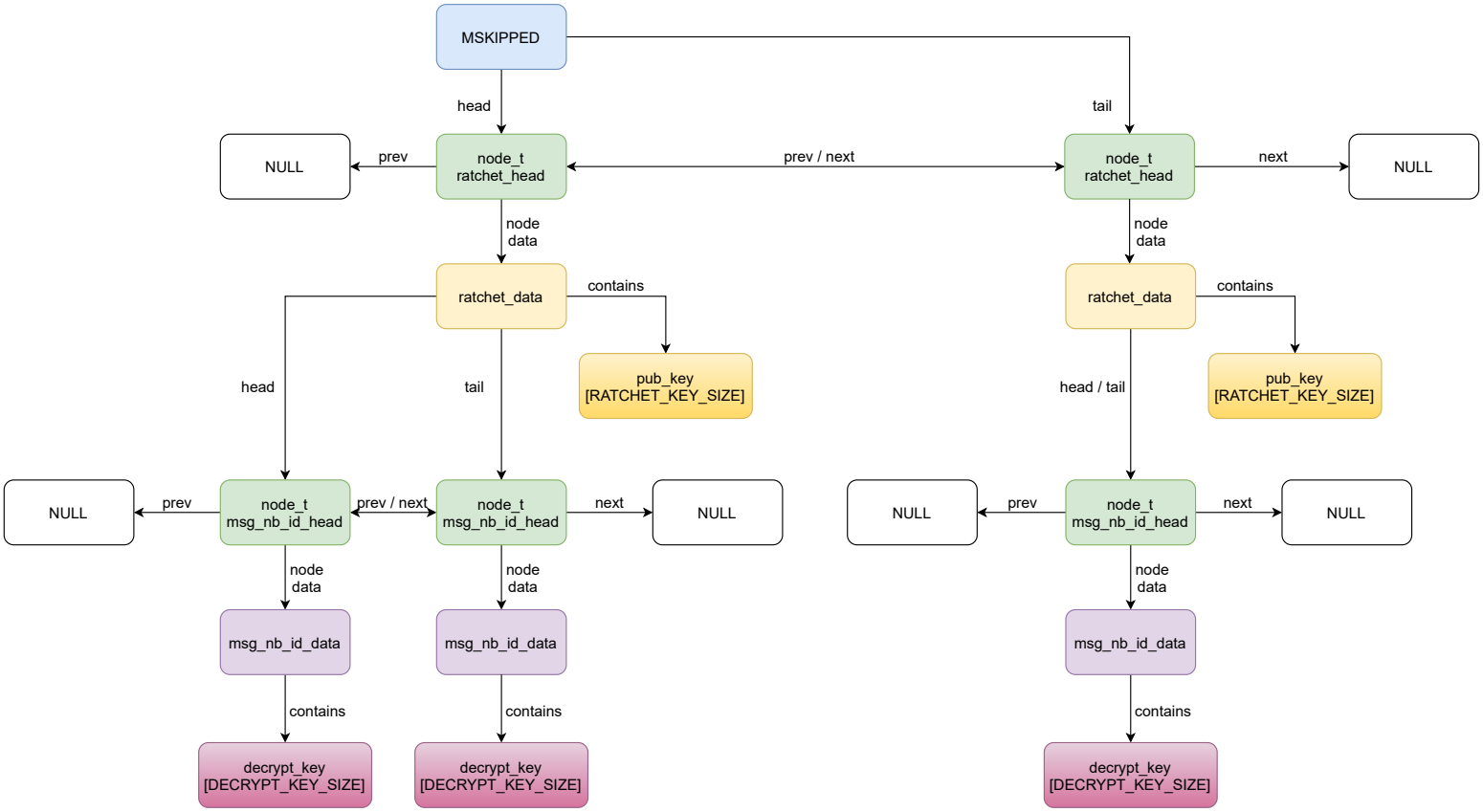
A.1 Authenticated Encryption with Associated Data

The next scheme roughly represents the different steps needed in the authenticated encryption with associated data (AEAD) utilized in this research. It simplifies the packet creation and exchange to stay as simple as possible. For the this reason, it does not take in account the header management between the two parties.


Sender
Receiver
msg key
from send chain
info
task specific
HKDF
w/ SHA256
salt set to
enough zeros
80 bytes
Auth. key
32 bytes
Enc. key
32 bytes
IV
16 bytes
Encrypt
AES256 in CBC
w/ PKCS#7
plaintext
associated
data (AD)
ciphertext
Concatenate
output to
authenticate
HMAC
w/ SHA256
HMAC
output
Concatenate
cipher +
HMAC(cipher+AD)
packet
msg key
from recv chain
info
task specific
salt set to
enough zeros
HKDF
w/ SHA256
80 bytes
IV
16 bytes
Enc. key
32 bytes
Auth. key
32 bytes
Decrypt
AES256 in CBC
w/ PKCS#7
plaintext
Concatenate
HMAC
w/ SHA256
associated
data (AD)
ciphertext
HMAC(cipher+AD)
Separate
HMAC(cipher+AD)
Verify
Signature
verified

A.2 MSKIPPED

The next graphic explains the structure created to store the messages keys for the out-of-order messages. This example represents the case where we stored three messages keys. Two with an ancient public key and one with an other public key. The green parts are nodes. They store a pointer to its direct neighbors and another pointer to the data of this node. The data are represented in yellow for the public keys and pink/purple for the message keys. The two upper data structures (blue and yellow) stores pointer to the head and the tail of the linked list of the layer below. The tail pointer eases the insertion and deletion of elements in the lists. They also store the number of message key stored in the layer below to know if we have to delete a node or not and to keep track of the total number of message keys stored. As explained in the documentation [13], we must have an upper bound to the number of message keys a party stores in order to avoid denial of services attacks.


MSKIPPED
head
tail
NULL
prev
node_t
ratchet_head
prev / next
node_t
ratchet_head
next
NULL
node
data
node
data
ratchet_data
contains
ratchet_data
contains
head
tail
pub_key
[RATCHET_KEY_SIZE]
head / tail
pub_key
[RATCHET_KEY_SIZE]
NULL
prev
node_t
msg_nb_id_head
prev / next
node_t
msg_nb_id_head
next
NULL
NULL
prev
node_t
msg_nb_id_head
next
NULL
node
data
node
data
node
data
msg_nb_id_data
msg_nb_id_data
msg_nb_id_data
contains
contains
contains
decrypt_key
[DECRYPT_KEY_SIZE]
decrypt_key
[DECRYPT_KEY_SIZE]
decrypt_key
[DECRYPT_KEY_SIZE]