POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

OpenThread overview and implementation of state-of-art attacks

Supervisors

Prof. Antonio J. DI SCALA

Company Supervisors

Marta FORNASIER

Guido BERTONI

Candidate

Davide CASALEGNO

05/2023

Summary

Internet of Things (IoT) is spreading everywhere and having an huge impact on the society by automating some actions and connecting everyday objects. IoT provides many improvements from the technological point of view, but has also some constraints due to the limited resources of the connected devices. The main application for IoT is home automation. In this field, the technology takes advantage of some protocols which are specifically designed for low-resources devices such as 6LoWPAN, 802.15.4 and Thread. This thesis focuses on studying the Thread protocol, a promising standard developed by Thread Group, that will become predominant in the IoT field. Thread is an IPv6-based mesh networking protocol developed for efficient communication between devices which have resource constraints. As Thread is a royalty-free but closed-documentation standard, Google Nest developed an open source implementation known as OpenThread. The goal of this work is to analyze the protocol and the underlying stacks it relies upon, provide a security overview about some critical phases of the protocol and implement some state-of-the-art attacks in a real-world scenario.

Acknowledgements

Grazie a tutti quelli che mi sono stati vicino durante questo percorso. Grazie alla mia famiglia e ai miei amici che mi hanno spronato a non mollare mai, anche quando le situazioni sembravano irrisolvibili. A Marta che, grazie ai suoi consigli e alle sue intuizioni, mi ha aiutato molto in questa ricerca. Al Professore Di Scala che è sempre stato disponibile per chiarimenti e approfondimenti molto validi.

OpenThread overview and implementation of state-of-the-art attacks

List of Figures VII

Acronyms IX

1 Introduction 1

2 Thread 3
- 2.1 Introduction to Thread . . . 3
- 2.2 Device types and roles . . . 4
- 2.3 Device types and roles during commissioning process . . . 5
- 2.4 Thread Network stack . . . 6
  - 2.4.1 IEEE 802.15.4 . . . 6
  - 2.4.2 Network Layer . . . 8
  - 2.4.3 Transport Layer . . . 13
  - 2.4.4 Message exchanged . . . 14
  - 2.4.5 Application Layer . . . 15
- 2.5 Commissioning Process . . . 15
  - 2.5.1 External Commissioning . . . 16
  - 2.5.2 On-mesh Commissioning . . . 20

3 OpenThread 23
- 3.1 Network Implementation . . . 25
- 3.2 Routing Layer Implementation . . . 25
- 3.3 DTLS . . . 25
- 3.4 J-PAKE . . . 26
  - 3.4.1 Key Establishment . . . 26

3.4.2 Zero-Knowledge Proof 27
3.4.3 J-PAKE over TLS 27
3.5 CoAP 28

4 Test Environment **29**
4.1 Border Router 29
4.2 Malicious Joiner 30
4.3 CLI commands 31
4.4 nRF 82540 SDK: compile and flash 33
4.5 Theoretical Attacks: 34
4.6 ATUSB configuration 35
4.7 802.15.4 sniffer configuration 36

5 Discussion **39**
5.1 Create the scenario 39
5.2 Attacks 39
5.2.1 Jamming Discovery Response 39
5.2.2 Flooding attack 40
5.2.3 Password Guessing Attack 41

6 Conclusion **42**

List of Figures

2.1 Different Roles for a Thread Device 5
2.2 Thread Stack 6
2.3 802.15.4 Frame 7
2.4 IPv6 datagram format 10
2.5 DTLS Exchange for Commissioning 15
2.6 Petitioning in an external scenario 17
2.7 Border Router is not the Joiner Router 18
2.8 Border Router is the Joiner Router 19
2.9 Petitioning in an on-mesh scenario 20
2.10 Joiner-Joiner Router-Commissioner 21
2.11 Joiner-JoinerRouter/Commissioner 22

3.1 OpenThread structure 24

4.1 nRF connect extension for visual studio code 33
4.2 802.15.4 ATUSB 35
4.3 transition state of the ATUSB 36
4.4 Timing for RX_START 36
4.5 nRF52840 Dongle for the sniffer [24] 37
4.6 nrfConnect for flashing the dongle 37

5.1 Commissioner starts the process 40
5.2 Joiner tries to join but it doesn't receive any answer 40
5.3 Wireshark result about the jamming 40

Acronyms

AI
: artificial intelligence

MAC
: medium access control

MIC
: Message Integrity Code

IoT
: Internet of things

IP
: Internet Protocol

6LoWPAN
: IPv6 over Low-Power Wireless Personal Area Networks

FTD
: Full Thread Device

MTD
: Minimal Thread Device

PDU
: Protocol Data Unit

CSMA-CA
: Carrier Sense Multiple Access-Collision Avoidance

RIP

Routing Information Protocol

RLOC

Routing Locator

ALOC

Anycast Locator

LLA

Link Local Address

ULA

Unique Local Address

GUA

Global Unicast Address

ICMP

Internet Control Message Protocol

PAN

Personal Area Network

MTU

Maximum Trasmission Unit

NHC

Next Header Compression

MLE

Mesh Link Establishment

RSSI

Received Signal Strength Indicator

TLV

Type-Length-Value

UDP

User Datagram Protocol

DTLS

Datagram Transport Layer Security

PSKc

Pre Shared Key for commissioner

PSKd

Pre Shared Key for device

CoAP

Constrained Application

J-PAKE

Password Authenticated Key Exchange by Juggling

OpenThread overview and implementation of state-of-the-art attacks

Chapter 1

Introduction

IoT (Internet of Things) is an emerging area and it will become very important in few years. The presence of low-battery system around everyone's houses is constantly increasing day by day. A discussion about the security of these devices must be done in order to be sure that everything works fine. IoT devices are mainly wireless device that, for example, can be added in an home and they are designed for the purpose of automating everyday tasks.

IoT device provides several benefit to their end users but they also raise serious security concerns because they interact with the physical world and they can cause some serious trouble if they are misconfigured since they can be used to implement sensors or actuators.

However,since IoT device can be used to unlock doors, turn on the lights,control sensible data or getting involved in some other critical activities, it can become very dangerous for the end users if the security of the various communication protocols have some vulnerabilities.

The two main communication protocols that are used in the low-power wireless personal area network are Zigbee[1] and Thread[2]. Amazon, Apple, Google and the Zigbee Alliance in 2019 announced a new project for the development of an IP-based smart home connectivity standard that would increase compatibility among IoT devices. This unifying standard was known as Project Connected Home over IP (CHIP) and it was re-branded as Matter in may 2021.

It is now a standard and it can use as underlying networking protocol Thread since it is an application-agnostic protocol that enables IPv6 based low power wireless mesh networking. The heterogeneity of the devices that are present inside a home can be an important factor given the different aim they have to satisfy, in fact some devices like smart motion sensors may require a longer battery life rather than a smart video-surveillance cameras that may require an high-data-rate connection. That's why IoT architectures often are build on top of a gateway that supports multiple communication protocol in order to reach different networks that otherwise

would be disconnected. Since Thread is application-agnostic, manufacturers of Thread products have the possibility to choose from different application layers to enable connectivity between different networks. The open-source implementation of Thread protocol is called OpenThread[3], it is available on github and it can be used for free for testing purposes. In this work we will focus on analyzing Thread protocol and the networking layer on which it relies, analyze its security, and implement a number of known state-of-art attacks.

Chapter 2

Thread

2.1 Introduction to Thread

This chapter will take a look at the Thread protocol by providing an overview both from a topological point of view, thus of the interaction between various devices, and a technical analysis of the stack on which Thread relies. The last part of the chapter is devoted to the commissioning process of getting a new device authenticated within the network. Thread [2] is a mesh networking protocol used in low-power and low-latency context. Thread aims to solve challenges such as ease of installation, interoperability, security and reliability. It is based on pre-existing technologies such as IPv6, 6LoWPAN and IEEE 802.15.4.

The main goals that Thread wants to reach are:

- Isolation of the Thread networks that is crucial for the security because a device which wants to connect to the network must authenticate himself thanks to a specific process called commissioning. Each device must authenticate himself and each communication is encrypted.
- Reliability: each device that is part of a thread network is supposed to do his job for a specified time and even if something unexpected happens. The network is auto-configuring and self-healing in order to avoid a single point of failure.For example if part of the network collapse or partitions it must be able to react in the correct way.
- Interoperability: thanks to the application-agnostic concepts different devices with different specifications could operate properly and understands each other.
- Scalability: there are limits in the number of devices that can be connected in a thread network. There could be up to 32 router (a thread device with

routing capability) and for each router up to 511 end devices. The network tries to keep the number of routers between 16 and 32 and that's why if the number of connected device is below 16, every REED (a role a device can assume) promotes himself to Router.

2.2 Device types and roles

There are many devices types and roles that a Thread device can play. [4]

A Thread device can be a Full Thread Device (FTD) or a Minimal Thread Device (MTD). MTDs have lower capabilities and significant resource constraints and that's why they cannot assume every role in a Thread scenario.

A FTD is the more capable device in this context and it keeps his radio always on. A FTD can serve in a variety of role such as can be a Thread Router which is able to forward packets on the network, to offers secure commissioning services for devices attempting to join the network and keeps its transceiver enabled at all times. It subscribes to the all-routers multicast addresses and maintains IPv6 address mapping. A FTD can also operate as a Thread Leader (role defined below) which has more responsibilities due to the management of the network. If the Thread Leader disconnects at some point, another Thread Router will be elected to become the Thread Leader. If a network partitions due to the lost of connectivity between devices, each partition becomes a network with its own Thread Leader that can be chosen by some other Thread Routers inside the partition. A FTD can also be a Thread Router-Eligible End Device, which doesn't provide routing services but can be promoted to a Thread Router thanks to the Thread Leader if the required conditions are met. If a FTD is not able to operate as a Thread Router, then it act as Thread Full End Device which communicates primarily with a single Router, it doesn't forward packets for other network devices and it can disable its transceiver to reduce power. There is a Parent-Child relationship between a Router and an End Device of order 1 to 1. The Router is always the parent, the End-Device is the child.

A MTD doesn't subscribes to the all-routers multicast address and forwards all messages to its Parents. It could operate as a Thread Sleepy End Device in order to save energy disabling the transceiver when it is idle. Alternatively an MTD could operate as a Thread Minimal End Device that left its radio enabled even when it is idle.

2.3 Device types and roles during commissioning process

The commissioning process was designed to add a new device to an existing network. During the commissioning process, an OpenThread device could act as:

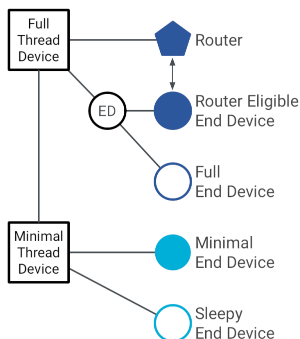

Figure 2.1: Different Roles for a Thread Device

- Thread Leader: controls the network configuration and allows Commissioner Candidate to become Commissioner and that he is the only one active in the network thanks to petitioning procedure.
- Commissioner Candidate: role for a device that has the potential to become the Commissioner but has not be designed yet.
- Commissioner : the device who authenticates the Joiner and provide it network credentials that are needed to join the network. It could be part of the network or it can be outside.
- Joiner: Role for a new device that wants to join the Thread Network. It exchanges messages with the Commissioner through the directly connected Joiner Router that serves as a proxy.
- Joiner Router: Role for a router device which is one hop away form the joiner device in the Thread network and is the only device connected with the joiner. It responds to the Discovery Request of the Joiner. Moreover, when chosen by the Joiner, it passes subsequent communication in a secure manner.
- Border router : Role for a device that forwards data between a Thread network and a non-Thread network. For this purpose it is equipped with at least two interfaces, for example Wi-Fi,Ethernet or other in addition to Thread. The Border Router can also be an interface for the Commissioner. The Border

Router is usually combined with the Border Agent function, which accepts petitions from the Commissioner Candidate. It also send out commissioning messages between Thread Network and a Commissioner. A node can be both a Commissioner and a border agent if the commissioner role is on-mesh.

2.4 Thread Network stack

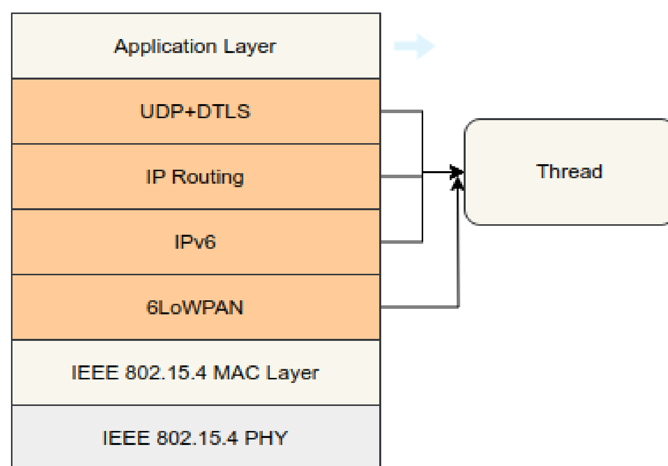

Figure 2.2: Thread Stack

As mentioned before multiple Thread device could form a Thread Network which is IPv6-based, some of them may be Routers or Border Router. Each Router has its own 16 bit address assigned by the Leader while other devices have address derived from their parent Router which is 64-bit that it is used at Medium Access Control Level for IPv6 addressing purposes. Additionally, IPv6 Unique Local Addresses and Global Unicast Addresses may be present on each device. The 6LoWPAN adaptation layer, which is located above the MAC layer and stands for IPv6 over Low-Power Wireless Personal Area Network, is how Thread devices deal with IPv6 packets. More specifically, the focal point about 6LowPAN adaptation is packet fragmentation and header compression. Thread network relies on IEEE 802.15.4 standard for the low level such as the Physical and link ones.

2.4.1 IEEE 802.15.4

IEEE 802.15.4 is the Thread's chosen standard for the low levels of the Thread networks.This standard is designed for low-rate application that fit perfectly in a IoT scenario. Link level security is mandatory in a Thread Network. This standard clarify how Physical layer and MAC layer must be managed. The protocol uses

a 127 bytes size frame at the Physical level to reduce the possibility of Bit Error Rate when using energy constrained device. Depending on the security settings and addressing type, the MAC layer payload might be as little as 88 bytes.

Frame Format IEEE 802.15.4 frames defined by the standards could be:

- Physical Layer Packet Structure: Each PDU(Protocol Data Unit) contains a synchronization header (preamble plus start of packet delimiter), a PHY header to indicate the packet length, and the payload that can be from 2 to 127 bytes. The PHY Header is made up by: 32-bit preamble that is designed for the acquisition of symbol and chip timing,8 bits for the Start Packet Delimiter and 8 bits for the length Field.

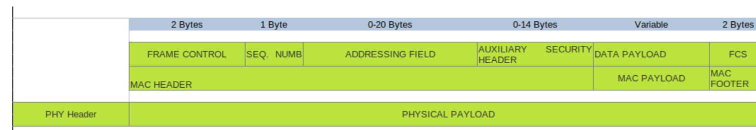

Figure 2.3: 802.15.4 Frame

Physical Layer

The physical layer must take care about some low level functionalities such as:

- Activation and deactivation of the radio transceiver in order to communicate. The radio transceiver could be in transmitting, receiving or sleeping mode.
- Channel frequency selection. IEEE 802.15.4 PHY works at 2.4 GHz band which is made up by different channels. The top ten are already in used, so one from 15 different channels that are numbered from #11 to #26 must be chosen to communicate.
- Physical Data transmission and reception that is the main task for this layer and it uses techniques like modulation and spreading.

MAC Layer

The main role of this layer is message handling and congestion control. This layer uses a CSMA-CA (Carrier Sense Multiple Access-Collision Avoidance) in order to allow multiple access for devices avoiding in advance collisions. Authentication , replay protections and encryption are used to grant confidentiality and reliability in end-to-end transmission. The layer supports two modes to operate: beacon mode

and non beacon mode. The one used by Thread protocol is the non-beacon mode where data are transmitted using unslotted CSMA/CA. No beacons are generated, so the Physical layer can provide higher scalability but cannot provide reliability of the packets delivered.

2.4.2 Network Layer

IPv6

IPv6 is a best-effort networking protocol and it is the most recent version of Internet Protocol developed by IETF (Internet Engineering Task Force) and is supported by Thread. It is designed to supply IP addressing and it uses DHCPv6 for router address assignment. It is considered as a successor of the classic IPv4 protocol and it was developed because the 32 bit IPv4 address space was beginning to be used up. There are different type of addresses which have different purposes:

- Unicast: it identifies a single interface represented by a RLOC (Routing Locator) How is RLOC generated? Each device has a RouterID and a ChildID and the combination of them identifies uniquely each device. Since a router is not a child, his ChildID will be always 0. They are used to create RLOC16 that is part of the Interface Identifier, which is the last 64 bits of the IPv6 address.[5] Different types of unicast addresses can be assigned to a device:
  - Link Local Address (LLA): they are used in a segment of the network and all interfaces can be reached by a single radio transmission but they cannot be routed.
  - Mesh-Local EID : it is also called Unique Local Address (ULA),it can be routed, but only within one routing domain. They are used for services that cannot be publicly accessible since they don't change as the topology changes. The prefix for this kind of addresses is always fd00::/8.
  - Global Unicast Address (GUA): it is used to identify an interface on the global scope beyond the Thread network. It is a public address and it has always 2000::/3 as prefix.
- Anycast: It is used to route Traffic to a Thread interface when the RLOC of the destination is unknown.It uses Anycast Locator (ALOC) to identify the location of multiple interfaces in a partition of the network thanks to RLOC lookup.The last 16 bits of an ALOC, called ALOC16 constitute the type of ALOC, it is in the format of 0xFCXX.
- Multicast: it identifies a group of hosts that share the same address. In a Thread network there are reserved addresses for multicast use.

Thread devices setup one or more ULA or GUA addresses. The 64-most significant bits of an IPv6 address identify the network instead the leftovers bits identify uniquely a device in the network.The most significant bits are also called prefix and they are usually 64. Each device uses its Extended MAC addess to obtain an unique identifier to configure its link local IPv6 address with the prefix FE80::0/64 ICMPv6, which is the adapted version of ICMP for IPv6 network, is used for watching over the network, error handling and signaling and for Neighbour Discovery Protocol with the echo request and echo reply that replaces the ARP protocol.

IPv6 datagram format:

The format of the IPv6 datagram is shown in figure and a lot of changes were added to exceed IPv4 problems.

- Expanded addressing capabilities: the size of the addresses is increased from 32 to 128 bits. This is enough to guarantee that the world will not run out of addresses.
- A fixed length of 40 byte header: a lot of IPv4 fields have been dropped and the fixed size of the header ensures a faster processing time of the datagram for any Router.
- flow labeling: now it is possible to label packets that are part of the same flow (e.g. video and audio packets.

There are different fields:

- Version: This field identifies the protocol version used and it is 4-bit long.
- Traffic Class: It is used to give priority to certain datagrams that belong to a specific flow or to ensure priority to specific application's datagrams rather than other. It is 8-bit long.
- Flow label: it is used to recognize a specific flow.
- Payload length: It is a field of 16-bit length and it is handled as an unsigned integer that give the size of the datagram following the fixed-length (40 bytes) of the header.
- Next header: this field point out the protocol to which the content of this datagram will be supplied.
- Hop Limit: this field indicates how many hop the datagram has done. Each router will decrement by one its value and if it reach 0, the datagram will be thrown away.

- Addresses: the two fields that contains the source and the destination addresses.
- Data: this is the payload of the packet that at the end of the communication will be go to the protocol described by the Next Header field.

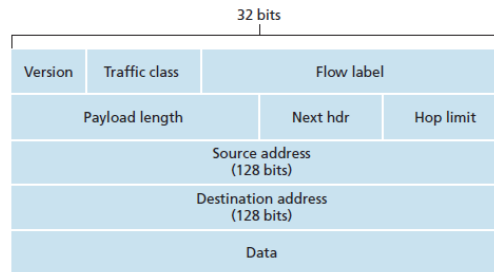

Figure 2.4: IPv6 datagram format

6LoWPAN

It is the core concept of the whole networking stack since it was born with the aim to standardize technical specifications about how to send and receive IPv6 packets over 802.15.4 networks. 802.15.4 standard mark out two possible addresses: Extended IEEE EUI-64 bit addresses that are unique globally and SHORT-16 bit addresses that are unique only in a chosen PAN. Both types are supported by 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks). 6LoWPAN assumes that a single PAN is mapped inside a specific IPv6 link, this implies a common prefix for devices that belong to a network. IPv6 supports multicast communication that is not provided by IEEE 802.15.4 standard, on the other hand the IEEE 802.15.4 uses massive broadcast communication that is not supported by IPv6. That's why 6LoWPAN can be seen as an adaptive layer since it can allow the integration between the two layers (IPv6 and IEEE 802.15.4) so far as broadcast frames can reach only devices that share a PAN mapped inside a specific link IPv6.

This allows IPv6 multicast packets to be encapsulated in IEEE 802.15.4 frames.

The main limitation is the different size of the two packets: the typical size of IPv6 PDU (Protocol Data Unit) is 1280 bytes and it must be carried in a 802.15.4 PHY frame which has an MTU (Maximum Transmission Unit) of 127 byte.[6] This issue is solved thanks to a 6LoWPAN property which is called packet fragmentation and reassembly.The IPv6 MTU is fragmented at the sender and reassembled once it reaches his destination. Another important feature that 6LoWPAN provides is the header compression mechanism, which allows two devices that are communicating to send less byte over the air leading to a reduction of energy consumed by the devices.In order to do so, 6LoWPAN uses network wide known information in a

specific context like mesh local and global prefixes, these are controlled by the Leader who also is in charge of mapping prefixes and id-context as :

- **Mesh Local Prefix**
  - $FD00 :: /64 \rightarrow ContextId0$
- **Global Prefixes**
  - $2003 :: /64 \rightarrow ContextId2$
  - $2001 :: /64 \rightarrow ContextId1$

The last important feature that 6LoWPAN provides is the link layer packet forwarding.

- **6LoWPAN IPv6 Packet Encapsulation:** 6LoWPAN packets are built following the same path of the IPv6 packets and carry different headers for different functionalities. Each Header is introduced by a dispatch value that indicates the type of the header. Thread can use 3 different types of header:
  - Fragmentation Header Application
  - Compression Header
  - Mesh Header
- **6LoWPAN IPv6 Headers Compression:** header compression mechanisms to reduce the packet overhead is based on a set of hypothesis that are specific to 802.15.4 links. The two types that are used by Thread are IPHC (Improved Header Compression) for IPv6 Header Compression and NHC (Next Header Compression) for reducing the size of UDP Header.
- **Packet Fragmentation:** The size of the two packets is very different given the fact that 802.15.4 frame payload can not exceed 88 byte and the IPv6 MTU is 1280 bytes. Given the size of IPv6 Header (40 bytes) and for example UDP Header (8 bytes) the leftover for application layer payload can reach at most 40 byte. $(88Bytes - 40Bytes - 8Bytes = 40Bytes)$ Thanks to the header compression mechanism it can go up to 80 bytes, this is still not enough and a packet fragmentation method is needed. At the sender's side 6LoWPAN splits and sends one by one an IPv6 packet if it doesn't fit into a single 802.15.4 frame. Once all fragments reach the receiver's side they must be reassembled before being passed to the IPv6 layer. A timer is activated at the receiver's side in order to avoid the possibility to run out of memory. The first fragmented packet has a specific format and is made up of a preamble, the IPv6 compressed header and the first size of the IPv6 payload. The other

fragmented packets are used for containing the rest of the payload and they are enriched with specific fields that are used to rebuild the original IPv6 packets at the receiver. These fields are :

- Datagram size: it indicates the size of the packet.
- Datagram tag: used to ensure the uniqueness of the packet, it links all the fragments together.
- Datagram offset: It is used to reconstruct the packet and it indicates the offset in multiples of 8 bytes from the starting point of the original packet.

Routing and Network Connectivity

Thread network uses RIP (Routing information Protocol) algorithm to handle routing and connectivity between devices. The format of the messages used to exchange information between router is MLE (Mesh link Establishment). Routers occasionally share link cost information that is a measure of the RSSI (Received Signal Strength Indicator) thanks to MLE advertisement packets.

MLE Messages

MLE messages have a specific format and are transfered using a single-hop link local unicast and multicast addresses. They are used for mantaining routing costs between devices, configuring radio links, exchange configuration value among the network and detecting neighbors devices. The first byte of the message can be a 0 or a 255 to distinguish between secured and unsecured messages. The following part is the command itself if the message doesn't need security or a triple composed by an Header, the command and a short piece of information used for integrity of data called Message Integrity Code (MIC). The Command follow a specific pattern that is TLV (Type-Length-Value) in order to provide specific functionalities that are known to Thread. Examples of the main MLE messages can be found in the table below:

Type	Description
Link Request (0)	A request to establish a link with a parent
Link Accept (1)	The positive answer to a link Request
Link Accept and Request (2)	Accept a requested link and request a link with the sender of the request
Link Reject (3)	Negative answer to a request
Advertisement (4)	It is used to inform neighbors of network information and state
Discovery Request (16)	A message used to discover network
Discovery Response (17)	Response to a Discovery Request

2.4.3 Transport Layer

Thread network is based on UDP (User Datagram Protocol) for communication between devices, but also TCP can be implemented as an application layer. UDP needs Internet Protocol (IPv6) as underlying protocol, it is a connection-less protocol that doesn't guarantee reordering and the re-transmission of the packets. It is quite efficient for lightweight application but it is considered less reliable than TCP.

DTLS

The current version used by Thread is DTLS 1.2 which is derived from TLS v1.2. Minor changes have been applied to DTLS compared to TLS. DTLS protocol has been developed to provide communication privacy functionalities to datagram protocol such as UDP. The main concepts are similar to TLS but DTLS is not an implementation of TLS over UDP. Since UDP doesn't guarantee the same principles of TCP (packets loss and packet reordering), DTLS has to make up for these weaknesses implementing low-energy strategies. Packets loss is handled with a local timer that can cause a re-transmission, unordered packets that reach destination are handled giving explicit sequence counter to each packet. If the received packet is the one expected, it will be processed, otherwise it will be queue up waiting for the previous ones. DTLS protocols is composed by three sub-protocols that are: the Handshake protocol, the alter protocol and the change chiper spec protocol.

Differences between TLS and DTLS

DTLS is intentionally very similar to TLS. The main difference is that TLS is based on top of TCP instead of UDP since it must relies on a transport protocol

that can provide no packet loss. Other differences are the explicit records since TLS splits data into chunks and DTLS uses records that must be reassembled at higher level. DTLS tolerates alterations and duplicates given the fact it is based on UDP. DTLS simply stops transmitting and no end signal is sent. TLS instead show the end of communication with a alert message. DTLS uses cookies to prevent IP spoofing, TLS allows the communications after an handshake making it difficult to spoof the IP address of the two actors. Another important difference is that stream ciphers are not allowed in DTLS but are quite common in TLS.

2.4.4 Message exchanged

The typical handshake is composed by different messages that are:

- Client Hello: The first message that is sent. Here the client propose some cipher-suite to perform the correct handshake. The cipher-suite proposed are typically a complex string which summarize all the algorithms, it includes key exchange algorithm,symmetric encryption algorithm and hash algoritm. Some example can be:
  - **TLS_ECJPAKE_WITH_AES_128_CCM_8**
  - **TLS_PSK_WITH_AES_128_CCM_8**
  - **TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256**

  It also specifies the supported group for the JPAKE exchange. The main group used is secp256r1
- Hello Verify Request: this packet is the first one sent by the server and it is used mainly to exchange cookie with the client. The cookie must be reposted in the next Client Hello.
- Client Hello (with cookie): this message is the same as the first client hello but with the addition of the previously received cookie.
- Server Hello: in this message the server answer back with the chosen Cipher-suite and enforces a session ID.
- Server Key Exchange:here the server specify the EC J-PAKE server parameters like the PubKey and the Schnorr signature.
- Server Hello Done: this message is used for to signal the end of this first part of the handshake, now the server waits for the client to tell it its parameters
- Client Key Exchange: here the client sends its EC J-PAKE parameters and the Schnorr Signature.

- Change Cipher Spec: this message is used to verify that nobody has altered the handshake.
- Encrypted Alert: this message is used to communicate the end of the handshake but it can also be used to signal problem and to close the communication channel.

The Handshake is used for commissioning and joining. The exchange of the messages can be seen in the images below.

ThreadCommissioning-JPAKE-DTLS-1.pcapng

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	483	Client Hello
2	1.305258	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	98	Hello Verify Request
3	1.316732	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	491	Client Hello
4	3.843320	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	499	Server Hello
5	3.845910	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	255	Server Key Exchange
6	3.846733	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	87	Server Hello Done
7	4.394968	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	252	Client Key Exchange
8	4.632689	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	76	Change Cipher Spec
9	4.632924	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	115	Encrypted Handshake Messa
10	5.906919	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	76	Change Cipher Spec
11	5.908021	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	115	Encrypted Handshake Messa
12	6.006600	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	115	Encrypted Handshake Messa
13	6.013250	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	129	Application Data
14	6.018072	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	129	Application Data
15	6.029483	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	115	Encrypted Handshake Messa
16	6.778019	2a03:39a0:1f:1004:b…	2a03:39a0:1f:1000:3…	DTLSv1…	93	Encrypted Alert
17	6.837024	2a03:39a0:1f:1000:3…	2a03:39a0:1f:1004:b…	DTLSv1…	93	Encrypted Alert

Figure 2.5: DTLS Exchange for Commissioning

2.4.5 Application Layer

This layer lives on top of all the Thread infrastructure, and since Thread is application agnostic it can be a custom application. The most widely used is Matter which is a standard for smart home that utilize as a routing stack Thread. The aim of Matter is to create a unified Standard for smart home device that can live in different ecosystem providing interoperability and reliability to all devices that are equipped with Matter.

Another example of application-layer protocol which is common to find is the Costrained Application Protocol (CoAP). One implementation of this protocol can be found in OpenThread.

2.5 Commissioning Process

Once the network has been created thanks to the GUI or in a manual way, it asks for a passphrase that is called COMMISSIONER CREDENTIAL and are used to generate the Pre-Shared Key for the Commissioner (PSKc). Commissioner credential is a user-defined string between 6 and 255 characters UTF-8 encoded.

This passphrase will be the pre-shared secret for the petitioning process. We have two phases:

- Petitioning
- Joining

Petitioning must happen before a joiner can enter the network and it is used to identify the only one authenticator node present inside the network.[7] Both phases are done using DTLS combined with Password-Authenticated Key Exchange by Juggling (J-PAKE) for key establishment. The Joining process uses a different low entropy key called Pre-Shared key for device (PSKd). This last key must have a very strictly format. The protocol used for the commissioning is called MeshCop (namespace used in the implementation of OpenThread) and it is based on CoAP and it conduct the petitioning process, management ad relay functions. The commissioning process could be in-mesh commissioning or external commissioning and different scenarios could arise.

2.5.1 External Commissioning

In external commissioning a device outside the network authenticates new devices on the Thread network. The different cases that we can meet are due to the system topology and the different roles that a node could play. With regard of topology and assuming that the external Commissioner is connected to the WLAN, we can have two cases :

- Border Router is not the Joiner Router with the Commissioner that is two degrees of separation from the Joiner.
- Border Router is the Joiner Router so the Commissioner will be one degree of separation from the joiner.

Since there are one or more degrees of separation, relay agents uses DTLS handshake.The relay agents will exist on the Joiner Router and the Border Router.

- **Petitioning in an external scenario:** An external commissioner has to petition the Thread Network to become the unique authorized Commissioner. The Commissioner candidate must perform an authentication handshake with the Border Router to become the only one Commissioner of the Network and to set up the secure Commissioning session. The process starts with the Border Router being conscious of the Commissioner Credential. This can be entered

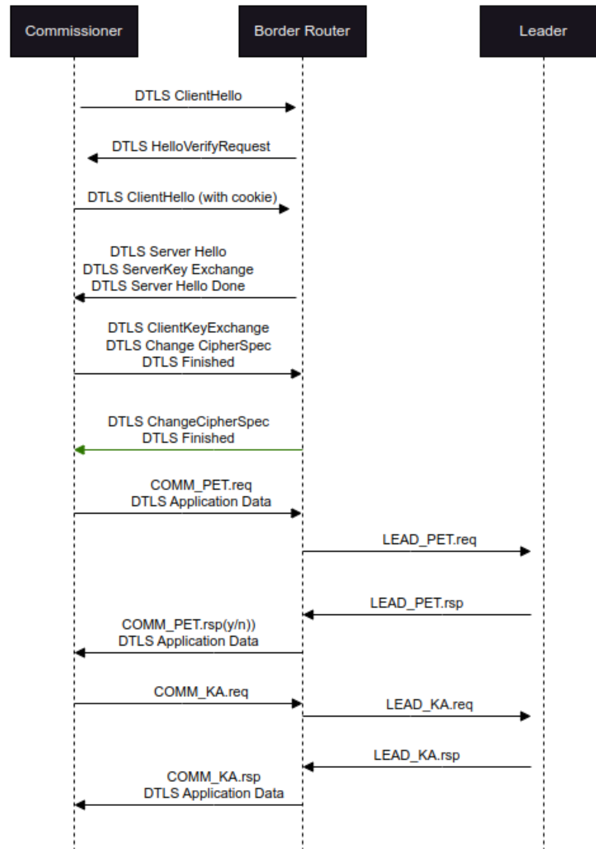

Figure 2.6: Petitioning in an external scenario

into the Border Router or into any trusted device and then sent to the Border Router, later the commissioner credential will be passed to the commissioner Candidate that will initiate the registration process and the DTLS handshake. If the handshake is successful, the Border Router will have authenticated the Commissioner Candidate based on knowledge of the shared commissioner Credential. The secure Commissioning session will remain opened between the authorized Commissioner and the Border Router and DTLS record protocol is uses for encryption and authentication based on keys derived from the master key established between the Commissioner and the Border Router as a result of the DTLS handshake. This commissioning session will be used for management and relay messages exchanged between the Commissioner and the Border Router.

- **Joining a Thread network with an external commissioner that is connected to the WLAN:** after the petitioning process have been taken

place, the secure session have been built up and it will be used to complete the joining process through to the commissioner, once the new device has received the network parameters, the session will be closed.There could be two different scenarios due to the fact that a node can assume different roles:

- **Border Router is not Joiner Router:** This is the most complex scenario since we have four different entities that need to communicate. There is the joiner, the commissioner, the border router and the Joiner router. In order to authenticate the traffic, there are three different paths of communication:
  * 1)Joiner to Joiner router point to point.
  * 2)Joiner Router to Border Router through the Thread Network.
  * 3)Border Router to Commissioner through WLAN.

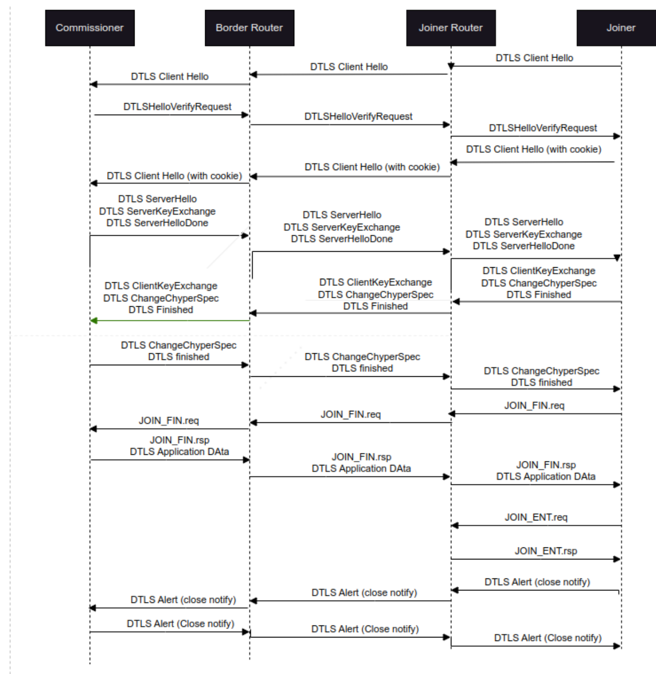

Figure 2.7: Border Router is not the Joiner Router

1. **Joiner to Joiner router point to point**: This is an insecure communication between Joiner and Joiner Router that is one-hop 802.15.4 radio link. The traffic is sent in clear without any form of integrity check. The traffic must be treated as unauthenticated. Normally the Thread network would be in "lock down" mode so any kind of unsecured 802.15.4 traffic would be ignored. When joining is

permitted, Joiner Router must take care carefully about that unsecured 802.15.4 traffic and identify it as authentication traffic. The first thing to do is the DTLS handshake between Joiner Router and Joiner. It is done on a specific UDP port and it is checked by a relay agent. The DTLS client handshake together with the address and the port details will be used also to reply to the joiner through the Joiner Router.

2. **Joiner Router to Border Router through the Thread Network:** This communication is entirely over the Thread network, and it is secured hop-by-hop at 802.15.4 MAC layer. The messages exchanged are hop-by-hop at 802.15.4 MAC layer. The messages exchanged are DTLS handshake along with address and port details.
3. **Border Router to Commissioner through WLAN:**This communication over the WLAN uses the secure Commissioning session that has been keep alive thanks to notifications. This is used to carry the DTLS handshake to and from the Commissioner and the messages are secured at the DTLS record layer.

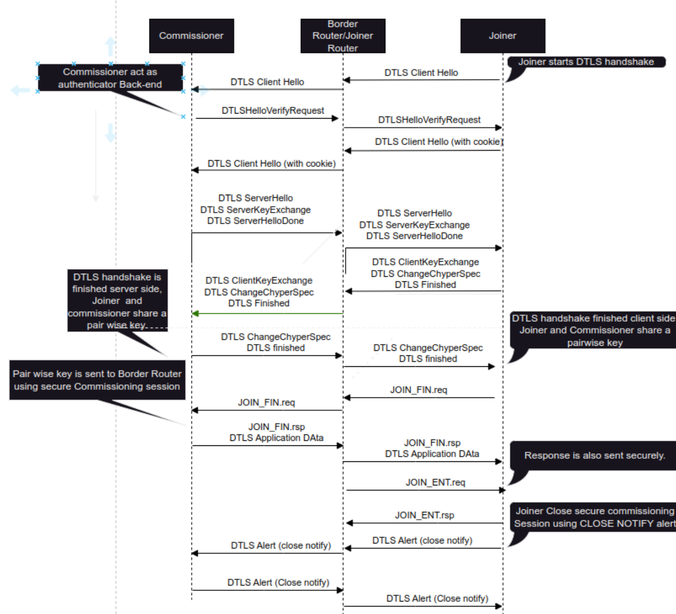

Figure 2.8: Border Router is the Joiner Router

– **Border Router is the Joiner Router:** Since the same node shares multiple roles it is not necessary to distinguish between different entities and the DTLS Handshake follow two specific paths previously described as Joiner to Border Router point to point and Border Router to Commissioner through WLAN.The figure below shows each step needed to commission a

new device in this scenario. The whole process starts from Joiner who is in charge of sending the first Client Hello. This is the first time that the Joiner Router meets the Joiner so it has to propagate the message to the Commissioner and it saves the port details in order to identify the return path. The Commissioner, once it has received the first client hello, it acts as an authenticator back-end that will perform the DTLS handshake which will be passed through the Border Router. DTLS handshake finished and then both parties share a pair-wise key.

2.5.2 On-mesh Commissioning

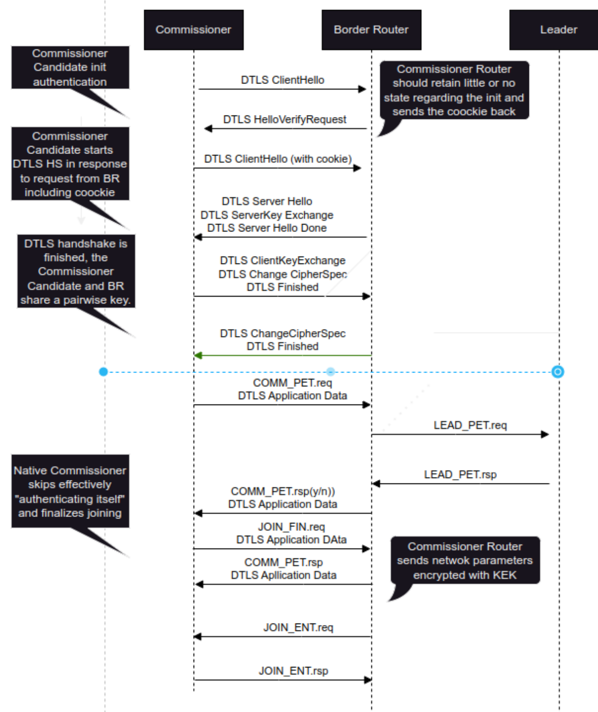

Figure 2.9: Petitioning in an on-mesh scenario

- **Petitioning in an on-mesh scenario:** the Commissioner Router is aware of the commissioner credential. This can be sent to the Commissioner Router by a trusted device or directly inserted into the device. Later the same credentials will be entered into the commissioner candidate that will initiate a DTLS handshake with the Commissioner Router. If the DTLS handshake succeed, the Commissioner candidate will be authenticated and it will join the network.

The leader of the network must guarantee that the Commissioner is the only one present in the network, otherwise the DTLS handshake fails.

- **Joining a new device in an on-mesh scenario:** Once the Commissioner has been established, new devices can join the network. The system topology can be various and considering only native commissioner we can identify two different scenarios:

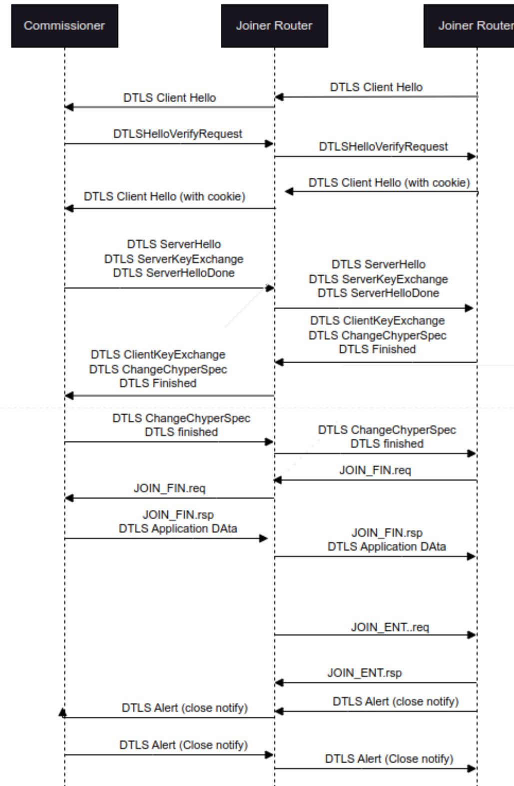

Figure 2.10: Joiner-Joiner Router-Commissioner

  - **Joiner router is not the commissioner:** in this case we have three actors that are: the Joiner,the commissioner and the joiner router. We have two separate paths for the authentication of the traffic. There is the communication between the Joiner and the Joiner router that is point to point and then there is the Joiner router that need to communicate with the Commissioner through the Thread Network. Joiner starts DTLS handshake and the joiner Router has no knowledge of the joiner at this point so it act as a pass-through by delegating the whole process to the commissioner. It will save the port and the path in order to be able to

answer back to the joiner. The commissioner will act as a authenticator back-end.

– **Joiner router is the commissioner:** in this scenario we have only two roles that are the joiner router which is also the Commissioner and the Joiner who talks directly to the it. The communication is point to point.The approach is client server and the protocol used is DTLS. The joiner starts DTLS handshake with the commissioner and once they have finished they share a pair-wise key securely.

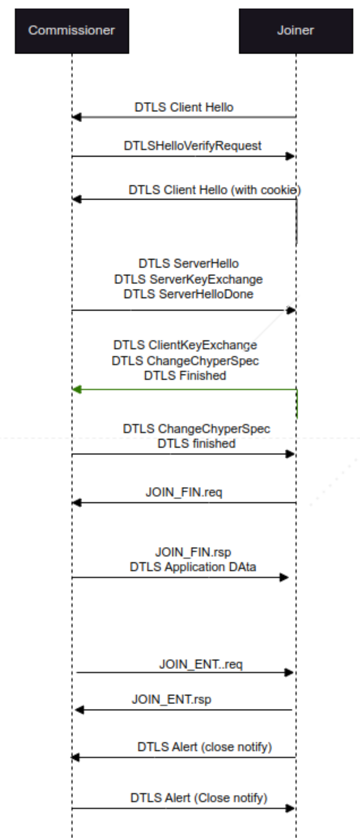

Figure 2.11: Joiner-JoinerRouter/Commissioner

Chapter 3

OpenThread

OpenThread was released by Google in 2016 as an open source implementation of the Thread Protocol and the whole code is available on github. It was developed keeping in mind that it must be highly portable and operative system agnostic. It can be used both on system-on-chip or as network co-processor. The main core is written in C++ 11 but the developers offers a lot of useful API, written in C, in order to interact in a friendly way with the Thread Stack. An helpful guide can be found online [8]. In OpenThread code, public data types and free functions should have **ot** prepended to their name. Variable and object names follow a convention. If an object is prepended with an **a** it will denote function parameter scope, instead an **s** is used for static scope. A data which has a **g** prepended to the name is used for object that have a global scope.[9]

The C++ code is divided in namespaces and the one which includes all the other is ot. It include other namespaces that will include as many definition of classes[10]. The one that have relevance for my work are:

- CLI
- CoAP
- Dns
- Ip6
- Lowpan
- MLE
- NetworkData

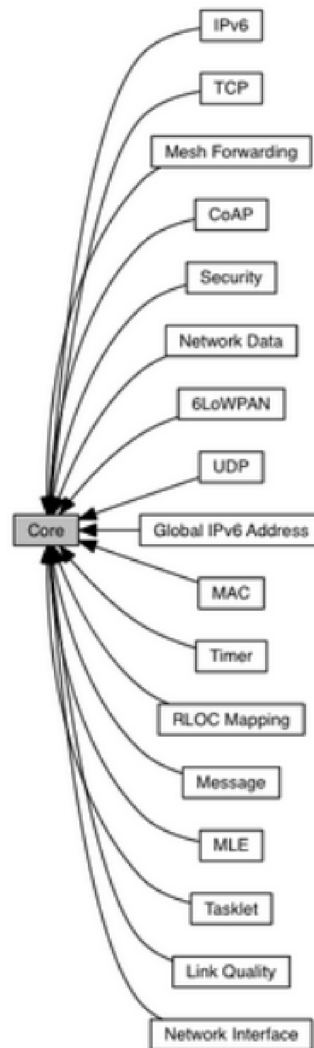

Figure 3.1: OpenThread structure

In order to add platform specific functions to the sample we must first declare them in the header file **./openthread/examples/platforms/openthread-system.h** and then we can implement them in the source file in **./src/src** directory. The code logic must be inside the main function that can be located **./openthread-/examples/apps/cli/main.c**[11] . To use an API which is already implemented we can include header to the application file and then call the chosen one.

3.1 Network Implementation

The namespace that is in charge of handling the networking part is the IPv6 one. All files related to this can be found in the folder, available in the github project, "openthread/src/net/". It includes various APIs that can be exploited and in turn implements a number of submodules that define various networking features such as ICMPv6,network interfaces, and IPv6.

In the file "../icmp6.cpp" we can find definiton for classes like Headers, ICMP. In the file "../ip6_address.cpp" there are the implementation of classes like Addresses, InterfaceIdentifier. In the file "../ip6.cpp" it can be found the definition of IP6 class and the implementation of methods that are used by the IP6 module.

3.2 Routing Layer Implementation

MLE is implemented in the "/src/core/thread" directory and is used to communicate Routing information and to maintain Routing tables. This information is used by the various actors within Thread such as the Routers and the Leader. This namespace also contains a header where various parameters are that is useful for setting the various parameters for Routing such as the maximum number of child that a router can have or the maximum number of Router that are allowed in a network.

3.3 DTLS

DTLS implementation is under the namespace MeshCop under the namespace ot. This module is in charge of handling DTLS connections and can be found in the folder "openthread/src/core/MeshCop" The main functions that an object of type DTLS are:

- DTLS::Dtls : This method instatiates a DTLS object.
- DTLS::Open : This method opens the DTLS socket.
- DTLS::Bind : This method binds this DTLS session to a UDP port.
- DTLS::Connect : This method establishes a DTLS session.
- DTLS::Close : This method closes the DTLS socket.

3.4 J-PAKE

J-PAKE is an algorithm of the family of PAKE (Password-Authenticated Key Exchange) and it is proven to provide perfect forward secrecy that is a property that ensure that the impairment of the private key will not affect the security of the derived keys, resistance to online and offline dictionary attacks. J-PAKE involves two actors and allows them to authenticate and establish a common secret. They both agree on group (G,g). Where G is a subgroup of $\mathbb{Z}_p^x$ with prime order q and g is the generator of G. They also share a low-entropy secret called s, that is exchanged with DTLS handshake. The shared secret cannot be 0 and must fall between 1 and q-1. It is articulated in two stages:

- Key Establishment Protocol: Negotiate a key for both actors in the communication
- Key Confirmation Protocol: both actors authenticate each other.

3.4.1 Key Establishment

It is done in two rounds

- **Round 1:** Alice chooses $x1 \in [0, q-1]$ and $x2 \in [1, q-1]$ while Bob chooses $x3 \in [0, q-1]$ and $x4 \in [1, q-1]$ Then Alice sends to Bob g^{x1}, g^{x2} and the Zero-knowledge proof for x_1 and x_2. The same is done by Bob who sends out g^{x3} , g^{x4} and the Zero-Knowledge proof for x3 and x4. When it finishes, they verify the Zero-Knowledge Proof and also check

  $$g^{x2} \neq 1, g^{x3} \neq 1$$

  .

- **Round 2:** Alice computes and sends out

  $$A = g^{(x1+x3+x4)*x2*s} \tag{3.1}$$

  and zero knowledge proof for x2*s. Bob computes and sends out

  $$B = g^{(x1+x2+x3)*x4*s} \tag{3.2}$$

  and the zero Knowledge proof for x4*s.

After Round 2, both of them compute K, Alice calculates

$$K = (B/g^{x2*x4*s})^{x2}$$

and Bob calculates

$$K = (A/g^{x2*x4*s})^{x4}$$

. In this way they both have

$$K = g^{(x1+x3)*x2*x4*s}$$

and they can derive a session key hashing K. The / must be intended as the multiplication with the inverse element. The inverse element can be calculated as

$$a/b = a * b^{-1}$$

3.4.2 Zero-Knowledge Proof

In order to produce a knowledge proof for the exponent sent we can use the Schnorr signature. J-PAKE relies on Schnorr Non-interactive Zero-Knowledge proof in order to achieve mutual authentication and key agreement based on a low entropy Pre-Shared Key that are the Pre-Shared Key for Commissioner (PSKc) for petitioning and Pre-Shared Key for Device (PSKd) for joining. We want to ensure the receiver that

$$X = g^x$$

and we send a tuple composed by

$$\langle ID, V = g^v, r = v - x * h \rangle$$

and the X. The ID is added for replay protection,

$$v \in \mathbb{Z}_q$$

and

$$h = H(g, V, X, ID)$$

and H is a secure function.The receiver must verify that

$$V = g^r * X^h$$

That is the proof that

$$X = g^x$$

3.4.3 J-PAKE over TLS

There is not a reliable infrastructure like PKI in the network so the certificate of the actors cannot be trusted but a confidential communication can be established even without the usage of certificates. Given the fact that there is a shared password between the two entities we can authenticate both of them using J-PAKE. If we starts from a TLS channel without authentication and then we authenticate both actors thanks to J-PAKE, we cannot avoid Man-In-The-Middle. To do so we have to mix TLS fingerprint in the shared secret.

3.5 CoAP

CoAP is an application-layer protocol that is implemented in OpenThread since it was developed for IoT. It is quite similar to HTTP but it was redesigned to perform better with device that are CPU and RAM constrained. The messages exchanged are smaller. CoAP can run on most devices that support UDP. CoAP makes use of two message types, request and responses, using a simple binary base header format. The base header may be followed by options that are exchanged in a TLV format. It is based on a client/server model and utilizes thread stack and DTLS to keep confidentiality for the communications. CoAP messages can be either confirmable or non-confirmable. Confirmable messages require an ACK, while non-confirmable don't.

Chapter 4

Test Environment

This chapter is about the environment needed to perform some state of art attacks. It will provide a list of the used hardware with a detailed installation guide. The original idea was to implement some attacks related to the commissioning phase where packets are not authenticated and can be manipulated. Thanks to Akestoridis' research [12] the two MLE command types that are not covered neither by MAC Security, neither by MLE security are Discovery Request and Discovery Response. This two messages cannot guarantee the integrity and the confidentiality since there is no MIC (Message integrity code).They are exchanged before the joining process. One of the attacks that was implement is about the online password guessing, as described by [12] Akestoridis. One of the possibility to implement password guessing is by using a malicious joiner, a device that is able to send request of joining that will try to guess the password <PSKd> which is chosen by the commissioner. The environment is made up by a Border Router which will form the Thread Network and will act as Thread Leader and as a Commissioner, an authentic joiner and a fake one that will try to join the network instead of the real one, an ATUSB flashed with attacks provided by Akestoridis's github, a sniffer to read the packets that are exchanged.

4.1 Border Router

The Border Router is composed by a Raspberry Pi 4 and an nRF52840 Dongle that will act as a Radio Co-Processor, in this way the Thread stack will run on the Raspberry Pi and the Dongle will act as a IEEE802.15.4 transceiver. The host needs to run two service that are otbr-agent and otbr-web. The Dongle must be flashed with the CLI sample code but instead of using as option OT_COMMISSIONER and OT_JOINER it will have OT_SRP_CLIENT and OT_EC-DSA option enabled. Once the nRF52840 Dongle has been flashed, it must be connected to the Raspberry

Pi and some packages must be installed thanks to nrfutil software.

4.2 Malicious Joiner

The latest version of nRFConnect SDK was used and thanks to the Visual Studio as IDE with the nRFConnect extension [13] and command line tools [14] I was able to set up the correct environment. I started coding from a sample cli project thanks to Zephyr OS [15] which is a Real Time Operation System, a flexible OS that allow to write linux application,using thread stack in an 802.15.4 context. It has already some functions and macro defined for OpenThread's project. The main function i used is the one to start the joining process which is an API called otJoinerStart() [10], it needs some parameters like the current instance and a callback to handle the joining process. the API is :

```
otError otJoinerStart(otInstance *aInstance, const char *aPskd,
    const char *aProvisioningUrl, const char *aVendorName,const char *
    aVendorModel,  const char *aVendorSwVersion,const char *
    aVendorData, otJoinerCallback aCallback, void *aContext)
```

[10] The callback which has the format:

```
void joiner_callback(otError joinErrorCode,void * passback);
```

should be used for error handling since the previous call can retrieve different types of error such as:

- OT_ERROR_NONE:Successfully started the Joiner role.
- OT_ERROR_BUSY:The previous attempt is still on-going.
- OT_ERROR_INVALID_ARGS: aPskd or aProvisioningUrl is invalid.
- OT_ERROR_INVALID_STATE: The IPv6 stack is not enabled or Thread stack is fully enabled

The Zephyr RTOS is multi threading so in order to be sure that the API is really executed, we must enclose its call between a mutex previously defined that ensure the atomicity of the call. The API related to the mutex are:

```
void openthread_api_mutex_lock(struct openthread_context *ot_context)
void openthread_api_mutex_unlock(struct openthread_context *
    ot_context)
```

which must use the context related to the Instance, we can retrieve a variable of type:

```
struct openthread_context * ctx= openthread_get_default_context();
```

Also a timer is needed to let the drivers work correctly and not keep busy the whole system so before any call we have to start it and wait till it expires.

A matrix filled up with the most used word that, following the specification of OpenThread about the format of a PSKd [16], must be all UpperCase Alphanumeric: 0-9, A-Z excluding I, O, Q, and Z, will be used in order to try different requests to join the network.

Another important change compared to the sample code of OpenThread is about MAC addresses that are randomly generated every time the device tries to exchange Discovery Request and Discovery Response. For this reason the code of the malicious joiner is modified in order to have a fixed MAC address which simplify the identification of the packets coming from the fake joiner. All the code can be found at the repository linked in [17]

4.3 CLI commands

There are a lot of useful commands provided by the OpenThread CLI [18] that i used in my work. Below the list of command used for this thesis work, with a brief explanation for each:

- **factoryreset**: this command brings to the default settings the device. The state of the device will be disabled.
- **ifconfig up|down**: which brings up|down the network interface and set the connectivity.
- **state**: this command display the state of the device.
- **ifconfig** : it shows the current state of the interface. It can be up or down.
- **ipaddr**: this command list the ip adresses that belong to an openthread device.
- **ipaddr add <ipAddress>**: it adds a new ipAddress to the network interface.
- **channel**: this command show the 802.15.4 channel that is currently in use or eventually to set it.

- **child table**: this command retrieve the table of the various child attached to an OpenThread device. It shows also more information which can be useful such as: RLOC16, Timeout, Age and extended MAC address of the various children attached.
- **dataset active**: this command provides the operational dataset which is active at the moment with a lot of useful information such as timestamp, the network key of the dataset, the network name, the extended pan id, the channel used by the dataset. If the -x option is added at the end of this command the whole dataset is formatted in an hexadecimal string which is more portable and can be useful to provide to a new device informations needed such as network key.
- **dataset init new**: it creates a new dataset.
- **eui64**: it shows the IEEE EUI64 assigned to the device.
- **netdata show**: it shows the network data received from the Leader.
- **networkkey <key>**: it sets the network key.

Some commands are used for the commissioning process such as:

- **commissioner start**: this command is used to start the petitioning process in order to promote the chosen device to a commissioner. There must be at most one commissioner in the whole network.
- **commissioner joiner add * <PSKd>**: this command is used to accept a new joiner which wants to enter the network and it also starts a timer, which is by default 120 seconds, that sets the maximum amount of time for the commissioning process. When the timer expires all the devices which haven't joined yet, are not more allowed to complete the joining process.
- **discover**: this command performs a MLE discovery operation that should reach all device in the network.

For the joiner who wants to complete the commissioning process these are the useful command:

- **joiner id**: it shows the joiner's ID.
- **joiner start <joining device credential>**: it starts the whole joining process and it attempts to start a DTLS handshake with the commissioner.
- **joiner state**: it shows the current state of the joiner.

  It can be: Idle, Discover,Connecting,Connected,Entrusted,Joined.

- **joiner stop**: it stop the joiner role.
- **extaddr**:this command is used to retrieve the extended MAC address or to set it to a chosen value.

4.4 nRF 82540 SDK: compile and flash

There are two main ways to build and flash the nRF82540 DK and both are useful. The first one is using the extension provided by the nRF connect which has a specific user friendly interface that shows the device and allow flashing the device.

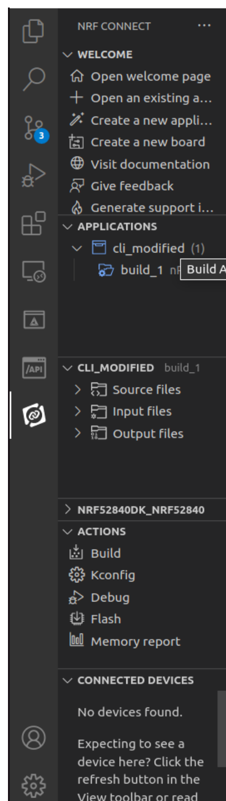

Figure 4.1: nRF connect extension for visual studio code

The other way is by CLI commands where we call a program (/ot-nrf52/script/build) to make the build of the project, setting the parameters specifying that it is not a MTD, that the joiner function is enabled.[11] See the following command :

```
$ ot-nrf52xx/script/build nrf52840 UART_trans -DOT_MTD=OFF -
  DOT_APP_RCP=OFF -DOT_RCP=OFF -DOT_JOINER=ON
```

then thanks to arm-none-eabi which is part of GNU Arm Embedded Toolchain [19] we convert it in hexadecimal, in order to be able to flash on the device

```
$ arm-none-eabi-objcopy -O ihex ot-cli-ftd ot-cli-ftd.hex
```

and then thanks to nrfjprog [19] we are able to flash the produced hex into the device making it work as expected. In the example below we use the ot-cli sample.

```
$ nrfjprog -f nrf52 -s serial_number --verify --chiperase --program
  ./ot-cli-ftd.hex --reset
```

4.5 Theoretical Attacks:

Jamming Attack: Jamming attack [20] is a well known attack in 802.15.4 network. In this scenario there is a jammer who launch the attack and he aims to degrade the network performance by inhibiting transmitted packets. Since one of the weak aspect of wireless network is their sensitivity to radio waves, a jamming attack make a variation of input signal bringing to a Different types of jamming attacks exist:

- **Constant Jammer:** the main concept is to transmit some noise made up by some random bits, in a continuos way without following a MAC protocol. In this case the jammer operates independently from the channel and the traffic on it.
- **Reactive Jammer:** In this scenario the jammer stay quiet and analyze the channel looking for transmission. The analisys stay passive if nothing happens but it will activate and starts to transmit noise if it detect some packets in order to corrupt them forcing the receiver to drop them and asking for the retransmission.
- **Random Jammer:** It switches between a dormant state and an active one. Both phases are timed with a random time to not let the defender predict a pattern. When the jammer is active it can act as a Constant jammer or a Reactive jammer.
- **Deceptive jammer:** in this scenario the Jammer keeps transmit packets so as to make believe that the channel is busy.

Spoofing attack: this attack is based on the fact that the attacker uses the address of another host, to take its place as a client (and hide its own actions) or

as a server.This attack if combined with the jamming one can be very useful inside a thread network since we can the spoofing of the acknowledgment of a device with the jamming of the packets previously sent by the device.[21]

4.6 ATUSB configuration

Figure 4.2: 802.15.4 ATUSB

ATUSB is based on a Atmega32U2 micro controller with an AT86RF231 transceiver which is high-accurate in timestamping by issuing an **RX_START** when the device is able to synchronize on a frame that is on the flies. When the transceiver wants to signal an interrupt, it toggles the IRQ (Interrupt Request) pin and the micro controller must notices it and thanks to the SPI(Serial Peripheral Interface), the micro controller reads the interrupt register associated and eventually clear it. In order to develop a jamming attack we can extend ISR (Interrupt Service Routine) and when we will receive a notification about the start of reception process, we will trigger a transaction to **PLL_ON** state, thanks to the command **FORCE_PLL_ON**, where we can start a transmission using SPI. [22]

Another important feature that the micro controller have is the possibility to issue an Address Match Interrupt, in this way the ATUSB can only start transmitting when a particular device matches the Address chosen.

The main functions used are

```
//transition into PLL_ON state
reg_write(REG_TRX_STATE, TRX_CMD_FORCE_PLL_ON);

//jam the received packet
spi_begin();
spi_send(AT86RF230_BUF_WRITE);
spi_send(jam_len);
spi_end();
```

The use of ATUSB is crucial for my work since it is used to try spoofing and jamming the thread network, i followed the instructions shown in the repository of Akestoridis[23].

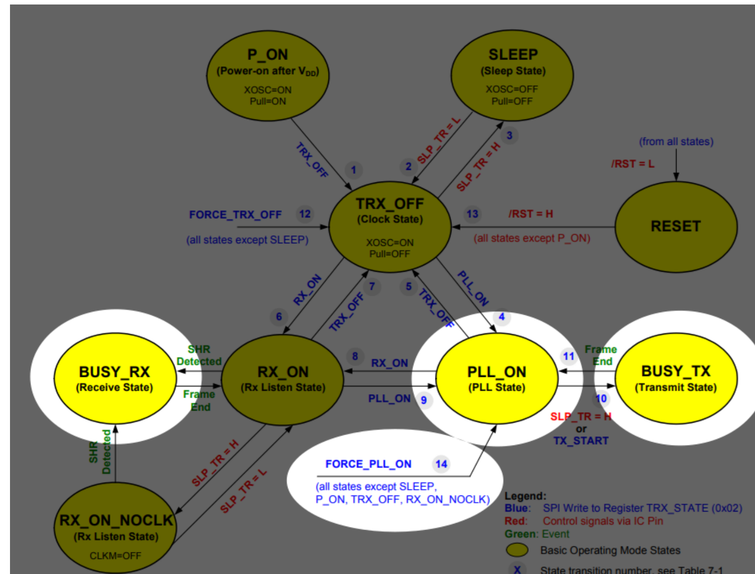

Figure 4.3: transition state of the ATUSB

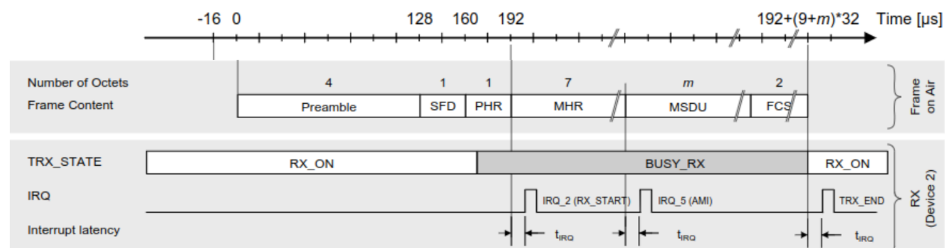

Figure 4.4: Timing for RX_START

4.7 802.15.4 sniffer configuration

The sniffer is configured based on a nRF52840 Dongle which is a RCP(Radio Co-Processor). This Dongle has no debugger included so we need to program it via bootloader which is provided by Nordic Semiconductor. As software we need :

- WireShark: [25]
- plugin for Wireshark and sniffer firmware.[26]
- Python3 [27]
- nrf Connect [13]

After having downloaded the repository, we have to copy the configuration for the sniffer written in pyhton in the extcap folder of wireshark. The other two files are

Figure 4.5: nRF52840 Dongle for the sniffer [24]

the hexadecimal file for the dongle or the development kit. They need to be flashed on the device thanks to the nRF Connect program. Wireshark must be setup to

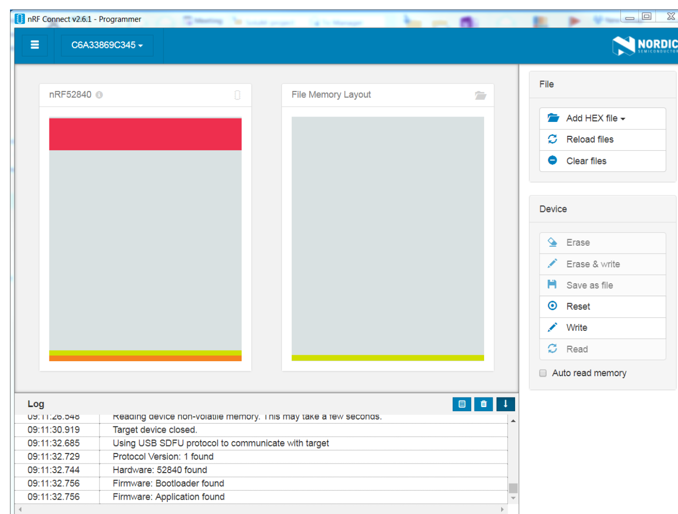

Figure 4.6: nrfConnect for flashing the dongle

sniff the 802.15.4 packets. In Edit $\rightarrow Preferences \rightarrow Protocols \rightarrow IEEE802.15.4$

the decryption key must be added and set to the network key of the thread scenario we want to analyze. we must set the context using the Mesh Local Prefix of the Thread network. Thread network uses as CoAP port the one at 61631 for the network management. In wireshark we have to set it up if it is needed.[28] The 6LoWPAN context must be set to ensure that the correct IPv6 addresses are displayed by wireshark. In order to decrypt the whole DTLS handshake the PSKd used for the process must be set with a proper format. Thanks to programs like

tr and **xxd** we can format in the correct way the PSKd and then Wireshark will display all the packets decrypted. The command to format the PSKd is:

echo <PSKd> |tr -d '\n' |xxd -ps -c 200.

The output of this command must be copied in edit $\rightarrow$ *preferences* $\rightarrow$ *protocol* $\rightarrow$ *DTLS* $\rightarrow$ *PSK*.

Chapter 5

Discussion

5.1 Create the scenario

The environment is composed by an OpenThread Border Router, some OpenThread Devices, an 802.15.4 sniffer and an ATUSB.

Forming the Thread Network The OTBR, which will act also as a Thread Leader, has to create the network,set the channel to the one chosen and commit the dataset to the network. The command used are:

- dataset init new.
- dataset channel 11.
- dataset commit active.

After these commands the IP interface must be enabled and the thread service must be started. the commands used to reach this condition are:

- ifconfig up.
- thread start.

5.2 Attacks

5.2.1 Jamming Discovery Response

The first attack that was tried is the Discovery Request jamming in order to avoid the joiner to complete the commissioning process. Before each commissioning process a Discovery Request and Response is used. Thanks to the attack #20 of

the repository of the attacks provided by Akestoridis [12] it is possible to implement it. In this attack the main idea is to jam the Discovery Response. In the first part of the code we filter only the MAC packets that have a valid length, that are 802.15.4, which have the same pan id as the one chosen, which is a Discovery response. After having received the packets, we make a transitions to PLL_ON STATE and we start the jamming phase calculating the length of the jamming packet and jamming the received on.

The Commissioner will set up the interface and starts the commissioning process:

```
>
> ifconfig up
Done
> joiner start J01NME
Done
> Join failed [NotFound]
```

Figure 5.1: Commissioner starts the process

The joiner will attempt to initiate a DTLS Handshake but it will not receive answers.

```
>
> ifconfig up
Done
> joiner start J01NME
Done
> Join failed [NotFound]
```

Figure 5.2: Joiner tries to join but it doesn't receive any answer

The results can be seen in Wireshark where we can see that the discovery request is correctly sent but there is no associated discovery response:

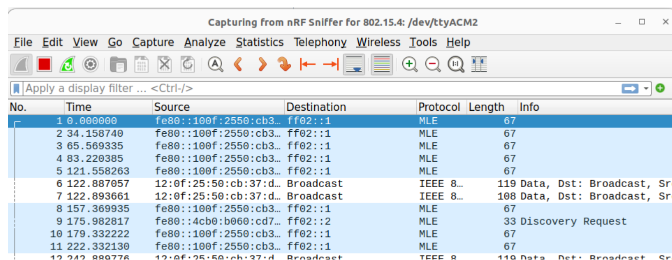

Figure 5.3: Wireshark result about the jamming

5.2.2 Flooding attack

Flooding is one of the first try that was implemented. A flood attack where the malicious joiner keep busy the commissioner and it avoid the possibility to

complete a correct handshake between an authentic Joiner and the Commissioner Candidate. This is due to the malicious joiner that tries to send an huge amount of Discovery Request and initiates a lot of commissioning process. It keeps open this connections and overload completely the Commissioner. This is due to the algorithm chosen since J-PAKE algorithm is quite expensive so the whole process cannot be verified for multiple requests. The commissioning process with the legitimate joiner sometimes won't complete and sometimes it takes a lot more time.

5.2.3 Password Guessing Attack

The last attack that was implemented is the password Guessing attack. This attack is more complex since the fake joiner will impersonate the authentic joiner. If the commissioner doesn't specify the 64-bit IEEE address, then the attacker could perform multiple password guesses during the commissioning process. The ATUSB could perform a jamming of the unsecured first fragment of the Client Hello of the authentic joiner which would prevent the reassembly of the message. Moreover since 6LoWPAN fragments rely on unsecured MAC acknowledgment, the ATUSB could spoof one after each selectively jammed first fragment to trick the authentic one into thinking it was received successfully. During this commissioning period the malicious joiner could tries to initiate multiple DTLS connections trying to guess the <PSKd> chosen for this authentication. The duration of a single commissioning period is about two minutes and then the whole process must be restarted. The jamming used in this case is a reactive jamming that is not active when the MAC address that tries to open a new connection is the one of the malicious joiner.

Chapter 6

Conclusion

In this thesis, a general study about Thread Protocol and its open-source implementation has been provided. Thread is built upon existing standards and can fulfill the requirements of low power, resilience, IP-based connectivity and security, to connect IoT devices. Furthermore, a real-world scenario has been implemented to observe its reaction to some possible attacks. With the setup created, it has been possible to reproduce some attacks that showed lack of security especially on the lower levels of Thread. However, the real feasibility of these attacks is not so straightforward, since their effects can be replicated only in a specific scenario, involving a border router that accepts connection without requiring a specific joiner's MAC. Working on Thread protocol for this thesis work, several obstacles have been met and, in my opinion, this technology cannot be considered completely user friendly due to the lack of documentations, not updated information and quite difficult information retrieval.

Despite it was overall feasible to use the Thread protocol in a real setup, there have been several problems related with flashing the devices, the documentation being quite messy and not easy to understand. In addition, a lot of time has been spent to understand the whole picture of Thread. A rework of the documentation should be done in order to help Thread users in the usage of the protocol. Another limitation of Thread, as well as many other IoT solutions, is the cost of the devices. The Thread products are quite expensive and creating a so called "smart-home" require a quite high investment, which cannot be easily reachable by everyone.

On the positive side, Thread has a bright future thanks to its versatility and scalability which will assure its adoption worldwide. In the near future the technology may become more mature and more people may be working on it, so there would be more applications and functionalities implemented. Thread could be the future of the mesh networking. In conclusion, this thesis work provides a first introductory overview to understand in a easy way how the Thread protocols works and what can be its pros and cons. As possible future work, students or Thread

users could deeper focus on the protocol continuing the work of processing and making the documentation more usable, as well as replicating or implementing some more attacks that were illustrated in this thesis.

Bibliography

[1] *Zigbee Alliance*. URL: https://csa-iot.org/resources/developer-resources/ (cit. on p. 1).

[2] *Thread Group*. URL: https://www.threadgroup.org/What-is-Thread/Thread-Benefits (cit. on pp. 1, 3).

[3] *openthread*. URL: https://openthread.io/ (cit. on p. 2).

[4] *Node Roles and Types*. URL: https://openthread.io/guides/thread-primer/node-roles-and-types (cit. on p. 4).

[5] *IPv6 Addressing*. URL: https://openthread.io/guides/thread-primer/ipv6-addressing (cit. on p. 8).

[6] *Thread Usage of 6LoWPAN*. URL: https://www.silabs.com/documents/public/white-papers/Thread-Usage-of-6LoWPAN.pdf (cit. on p. 10).

[7] *Commissioning Process*. URL: https://www.threadgroup.org/Portals/0/documents/support/CommissioningWhitePaper_658_2.pdf (cit. on p. 16).

[8] *OpenThread C API Reference*. URL: https://openthread.io/reference (cit. on p. 23).

[9] *OpenThread Style guide*. URL: https://github.com/openthread/openthread/blob/main/STYLE%5C_GUIDE.md (cit. on p. 23).

[10] *Specification*. URL: https://software-dl.ti.com/simplelink/esd/simplelink_cc13x2_sdk/2.30.00.45/exports/docs/thread/doxygen/openthread-docs-0.01.00/html/dd/d14/namespaceot_1_1Coap.html#details (cit. on pp. 23, 30).

[11] *Developing with Openthread API*. URL: https://openthread.io/codelabs/openthread-apis%5C#10 (cit. on pp. 24, 33).

[12] Dimitrios-Georgios Akestoridis, Vyas Sekar, and Patrick Tague. «On the security of Thread networks: Experimentation with OpenThread-enabled devices». In: *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2022, pp. 233–244 (cit. on pp. 29, 40).

[13] *nrf-connect*. URL: https://www.nordicsemi.com/Products/Development-software/nrf-connect-sdk (cit. on pp. 30, 36).

[14] *Command Line Tools*. URL: https://www.nordicsemi.com/Products/Development-tools/nrf-command-line-tools/download (cit. on p. 30).

[15] *Command Line Tools*. URL: https://zephyrproject.org/wp-content/uploads/sites/38/2023/04/Zephyr-Overview-2023Q1-Master.pdf (cit. on p. 30).

[16] *OpenThread Commissioning*. URL: https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/protocols/thread/overview/commissioning.html (cit. on p. 31).

[17] *my repo*. URL: https://github.com/secpat-dev/thesis_davide-casalegno (cit. on p. 31).

[18] *CLI Command References*. URL: https://openthread.io/reference/cli/commands (cit. on p. 31).

[19] *arm-none-eaby*. URL: https://developer.arm.com/downloads/-/gnu-rm (cit. on p. 34).

[20] *jamming*. URL: https://it.wikipedia.org/wiki/Jamming (cit. on p. 34).

[21] *Spoofing*. URL: https://it.wikipedia.org/wiki/Spoofing (cit. on p. 35).

[22] *Low-Cost ZigBee Selective Jamming*. URL: https://www.bastibl.net/reactive-zigbee-jamming/ (cit. on p. 35).

[23] *ATUSB Attacks*. URL: https://github.com/akestoridis/atusb-attacks (cit. on p. 35).

[24] *Dongle Images*. URL: https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dongle (cit. on p. 37).

[25] *Wireshark*. URL: https://www.wireshark.org/download.html (cit. on p. 36).

[26] *Wireshark's plugin for nRF52840*. URL: https://github.com/NordicSemiconductor/nRF-Sniffer-for-802.15.4/ (cit. on p. 36).

[27] *Python*. URL: https://www.python.org/downloads/ (cit. on p. 36).

[28] *Wireshark for nRF52840*. URL: https://openthread.io/guides/pyspinel/wireshark (cit. on p. 37).