UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

Comparative Study of Encryption Algorithms in Battery Powered Thread© Networks for Smart Homes

Permalink

https://escholarship.org/uc/item/8sz831cz

ISBN

9798273397316

Author

Tran, Simeon Alexander

Publication Date

2025-11-12

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

COMPARATIVE STUDY OF ENCRYPTION ALGORITHMS IN BATTERY POWERED THREAD© NETWORKS FOR SMART HOMES

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Simeon Tran

December 2025

The Thesis of Simeon Tran
is approved:

Professor David C. Harrison, Chair

Professor Alvaro Cardenas

Professor Christina Parsa

Peter Biehl
Vice President of Graduate Studies

NOTE

In August 2025, the National Institute of Standards and Technology (NIST) has standardized the algorithms defined in the ASCON cipher suite [1, 2]. However, this Master's Thesis was completed *before* NIST's standardization of ASCON. Instead, a third-party implementation of ASCON: LibAscon [3], was used in this thesis. Repeating this thesis research on the official implementations of the standardized ASCON algorithms, available in the official ASCON GitHub repository [4], is left for future work.

Table of Contents

List of Figures v

List of Tables vi

Abstract vii

Dedication viii

Acknowledgments ix

1 Introduction 1

2 Background 4
2.1 Advanced Encryption Standard (AES) 4
2.1.1 Overview 4
2.1.2 Encryption Algorithms 5
2.1.3 AES-CCM 6
2.2 ASCON 6
2.2.1 Overview 6
2.2.2 ASCON AEAD 8
2.2.3 Source Code 10
2.3 Constrained Application Protocol (CoAP) 11
2.4 Thread 14
2.4.1 Overview 14
2.4.2 Types of Devices & Network Topology 16
2.4.3 OpenThread 20
2.4.4 Thread in Smart Homes 20
2.5 Espressif IoT Development Framework (ESP-IDF) 22
2.5.1 Overview 22

2.5.2 ESP32-H2 . . . 22
2.5.3 ESP Thread Border Router . . . 23
2.5.4 ESP32 Deep Sleep . . . 25
2.6 Related Work . . . 26
2.6.1 ASCON in Real World Deployment Scenarios . . . 26
2.6.2 Performance of Thread . . . 27

3 Replacing the OpenThread Encryption Algorithm 28
3.1 Overview . . . 28
3.1.1 The nRF Packet Sniffer . . . 30
3.2 Removing AES-CCM . . . 31
3.2.1 Confirming Removal of AES-CCM . . . 33
3.3 Adding ASCON AEAD into OpenThread . . . 37
3.3.1 LibAscon Wrapper Functions . . . 37
3.3.2 The Encryption and Decryption Process . . . 38
3.3.3 Invoking LibAscon AEAD . . . 41
3.4 Challenges & Future Work . . . 43
3.4.1 A 16 Byte Tag is Too Big . . . 43
3.4.2 Commissioning Issue . . . 43
3.4.3 Outstanding Challenge for Future Work . . . 45

4 Evaluation 46
4.1 Correctness Tests . . . 47
4.1.1 Limitations . . . 48
4.2 Delay Tests . . . 48
4.3 Confirmable Tests . . . 52
4.4 Observe Tests . . . 56
4.5 Energy Consumption Tests . . . 61
4.6 Independent and Dependent Variables . . . 67
4.7 Test Environment . . . 68
4.7.1 Network Performance Experiments . . . 68
4.7.2 Energy Consumption Experiments . . . 70
4.8 Experimental Setup . . . 74
4.8.1 Network Performance Tests . . . 74
4.8.2 Energy Consumption Tests . . . 77

5 Results 79
5.1 Correctness Tests . . . 79
5.2 Network Performance Tests . . . 80
5.3 Energy Consumption Tests . . . 82
5.3.1 Waveform Analysis . . . 83

5.3.2 Differences in Battery Lifetime 85
5.3.3 Results 92

6 Conclusion **95**
6.1 Conclusion 95
6.2 Future Work 96

Appendices

A Challenges & Observations **97**

B Assertion Failures in Observe Experiments **126**

C Sleepy End Device (SED) Energy Usage Waveforms and Packet Captures **134**

Bibliography **140**

List of Figures

2.1 Differences between AES-128, AES-192, and AES-256 [27]. 5

2.2 The top and bottom diagrams display the encryption and decryption processes, respectively, under ASCON AEAD [18]. 8

2.3 Differences between ASCON-128a and ASCON-128 [18] . The variables p^a and p^b represent the number of permutations performed on the associated data, and the plaintext or ciphertext, respectively. 9

2.4 The values for the nonce, key, associated data, along with the block size and number of permutations used, for the ASCON-80pq cipher [18]. 10

2.5 A timing diagram showing how a retransmission occurs when an Acknowledgement for a Confirmable request is not received [38]. . 12

2.6 A timing diagram displaying a CoAP observe interaction between a client and a temperature sensor, which is running the CoAP server [38]. The measurement unit `Cel` corresponds to degrees Celsius. . 14

2.7 The Thread network stack [51]. 16

2.8 Device types in a Thread network from the OpenThread documentation [48]. 17

2.9 The topology of a typical smart home network under Thread [44]. A Thread smart home network has the ability to communicate to non-Thread devices that support the Matter application layer protocol. 21

2.10 The ESP32-H2 development kit [63]. The component labelled "ESP32-H2-MINI-1" is the System on a Chip (SoC) that can be attached as a part of a larger Microcontroller Unit (MCU) [64]. 23

2.11 Prebuilt ESP Thread Border [68,69]. The top and bottom modules are the ESP32-H2 and ESP32-S3 SoCs, respectively. 24

3.1 The nRF52840 dongle (left) [87] and the nRF52840 development kit (right) [88]. 30

3.2 The top figures display a User Datagram Protocol (UDP) packet that my ESP32 sent with the payload: "`hello_world`", as ciphertext encrypted by AES-CCM. The bottom figure shows payload after Wireshark has decrypted it. 35

3.3 A UDP packet with the payload "`hello_ucsc_plaintext_packet`" in plaintext. Note that Wireshark (unsuccessfully) decrypts this payload using AES-CCM, since all frames carrying application layer data will be encrypted in Thread. 36

3.4 Wireshark's inability to decrypt the Mesh Link Establishment (MLE) packets proved that the removal of MLE encryption was successful. 36

3.5 The format of a nonce as input to AES-CCM encryption, as described in 802.15.4-2006 [31]. 39

3.6 The serial monitor output of the commissioner. The commissioner successfully adds the joiner, but eventually removes the joiner after failing to communicate with it. 44

4.1 The floor plan of the single story ADU. The blue circle numbered 1 indicates the location of the delay client, and the green circle numbered 2 indicates the location of the delay server. The yellow circle with the number 3 indicates the location of the nRF52840 packet sniffer. 49

4.2 The floor plan of the single story ADU. The blue circle numbered 1 indicates the location of the Full Thread Device (FTD), and the orange circle numbered 2 indicates the location of the Border Router. The yellow circle numbered 3 indicates the location of the nRF52840 packet sniffer. 52

4.3 The floor plan of the ADU. The green circle numbered one corresponds to the location of the Border Router. The yellow circle numbered two corresponds to the location of the nRF52840 802.15.4 sniffer. The blue circles labelled 3, 4, and 5, corresponds to the locations of the sensor for the front door, the air quality monitor, and the window sensor, respectively. 61

4.4 The experimental testbed[1] used to run the Delay, Throughput, and Packet Loss experiments. The blue dotted lines indicate a Universal Serial Bus (USB) connection which enables only data transfer, the yellow dotted lines indicate a wired connection supplying Alternating Current (AC) power, the pink dotted lines indicate a USB connection which supports *both* data and power transfer, and the green dotted lines indicate an Ethernet connection. 69

4.5 A picture of the Power Profiler Kit II (PPK2) (top) and jumper cables (bottom) [127]. 71

4.6 The ESP32-H2 when the J5 jumper it was removed, and when covering the J5 header pins. The power-on LED is off when the J5 jumper is not plugged in, even though the device itself was still on (powered by USB). 72

4.7 Shown left and right are the setup of Configuration (1) and (2), respectively. 75

4.8 The locations of the FTD, nRF52840 802.15.4 packet sniffer, Border Router, and Delay Server in Configuration (3). The abbreviations "BR" and "DS" are short for "Border Router" and "Delay Server", respectively. The FTD remained in the same location throughout the evolution of the testbed, and operated as the delay client in the Delay experiments. 76

4.9 The experimental setup for the Energy Consumption experiments. SED (4) is not show in this picture, as it is hidden behind the wall past the door. 77

4.10 The Border Router, the SEDs, the nRF52840 802.15.4 sniffer, and the PPK2. 78

5.1 The results of the Delay experiments. 80

5.2 The results of the Throughput experiments. 81

5.3 The results of the Packet Loss Observe experiments. 82

5.4 The energy consumption waveform[2] for a SED during a single wakeup, along with the corresponding Wireshark packet capture. . 84

5.5 The average energy consumption of the SED in milliampere (mA), under the different encryption algorithms (along with when none is used). 85

5.6 The results of the Energy Consumption experiments. 92

List of Tables

4.1 Mappings between devices used in the Throughput Confirmable experiments to the corresponding commercial devices. 52

4.2 Mappings between the devices used in the experiments to the corresponding commercial devices. 62

List of Acronyms and Abbreviations

6LoWPAN IPv6 over Wireless Personal Access Networks.

AC Alternating Current.

ACK Acknowledgment.

AEAD Authenticated Encryption & Associated Data.

AES Advanced Encryption Standard.

AES-CCM Advanced Encryption Standard - Counter with Cipher Block Chaining-Message Authentication Code.

API Application Programming Interface.

BLE Bluetooth Low Energy.

CAESAR Competition for Authenticated Encryption: Security, Applicability, and Robustness.

CHIP Connected Home Over IP.

CLI Command Line Interface.

CoAP Constrained Application Protocol.

CPU Central Processing Unit.

CSL Coordinated Sampled Listening.

dBm Decibel-Minutes.

DTLS Datagram Transport Layer Security.

Enh-ACK Enhanced Acknowledgment.

ESP Espressif.

ESP-IDF Espressif IoT Development Framework.

FED Full End Device.

FTD Full Thread Device.

HMAC-SHA256 Hashed Message Authentication Code using SHA-256.

HTTP Hypertext Transfer Protocol.

ICMP Internet Control Message Protocol.

IEEE Institute of Electrical and Electronics Engineers.

IoT Internet of Things.

IPv6 Internet Protocol Version 6.

IRNAS Institute for Development of Advanced Applied Systems.

J-PAKE Password-Authenticated Key Exchange by Juggling.

KEK Key Exchanges Key.

LAN Local Area Network.

LoRa Long Range.

mA milliampere.

MAC Medium Access Control.

MAC Message Authentication Code.

mAh milliampere-hours.

MCU Microcontroller Unit.

MED Minimal End Device.

MeshCoP Mesh Commissioning Protocol.

MIC Message Integrity Counter.

MLE Mesh Link Establishment.

MQTT Message Queuing Telemetry Transport.

MTD Minimal Thread Device.

NAS Network Attached Storage.

NIST National Institute of Standards and Technology.

NVS Non-Volatile Storage.

PDR Packet Delivery Ratio.

PPK2 Power Profiler Kit II.

RCP Radio Co-Processor.

RDP Remote Desktop Protocol.

REED Router Eligible End Device.

REST Representational State Transfer.

RFC Request for Comment.

RTT Round Trip Time.

SDK Software Development Kit.

SED Sleepy End Device.

SoC System on a Chip.

SSED Synchronized Sleepy End Device.

TCP Transfer Control Protocol.

TLS Transport Layer Security.

TX Transmit.

uA microampere.

UART Universal Asynchronous Receiver/Transmitter.

UDP User Datagram Protocol.

URI Universal Resource Identifier.

URL Universal Resource Locator.

USB Universal Serial Bus.

UUID Universally Unique Identifier.

VPN Virtual Private Network.

Abstract

Comparative Study of Encryption Algorithms in Battery Powered Thread©

Networks for Smart Homes

by

Simeon Tran

In response to the security challenges inherent in the Internet of Things (IoT), in 2023, National Institute of Standards and Technology (NIST) has endorsed ASCON, a cipher suite designed to secure the communications between resource-constrained IoT devices. Thread, a popular wireless mesh protocol designed for smart homes and smart buildings, does not currently support ASCON. This thesis describes how OpenThread, the open source implementation of Thread, can be modified such that it is capable of using the encryption algorithms defined in the ASCON cipher suite. Compared to the original version of OpenThread, this thesis shows that the network performance and battery lifetime of smart home devices are not negatively impacted when OpenThread is secured by ASCON encryption. To the best of my knowledge, this thesis is the first to investigate the potential of ASCON in securing the communications of smart home devices operating under OpenThread.

To my father, mother, uncle, and sister,

for giving me much support and encouragement

throughout my life.

Acknowledgments

I would like to express my gratitude to my industry mentor, Jeff Cheang of SunTek Systems, for sparking my passion in cybersecurity, networks, and the Internet of Things. I want to additionally to thank fellow graduate students Mike Anjomani, Jonathan Castello, Daniel Sabo, Jonathan Casper, and Dustin Palea — they have given me much wisdom and life advice throughout my time in graduate school. Furthermore, would like to express my gratitude to graduate students Andrea David and Harikrishna Kuttivelil, who both additionally have given me much support and advice during my first two quarters as a graduate student.

Finally, I would like to thank engineer Michael Wei (who goes by the username `no2chem` on GitHub) for debugging the network time synchronization problems that I encountered on my ESP Thread Border Routers, and for taking the time to explain to me why those issues were occurring, and I want to extend my gratitude to the engineers at Digilent, Nordic Semiconductors, and Espressif, who took the time to answer the questions I posted within the online community forums of their respective companies. Furthermore, I additionally would like to thank the engineers at Espressif, Nordic Semiconductors, Digi, and Aqara, for sharing with me what they know about Thread, smart homes, and the Internet of Things during my time at Consumers Electronic Show (CES) 2024 convention.

Chapter 1

Introduction

From smart homes to smart cities [5–7], IoT technologies power the world that we live in today. Yet there exists a significant gap between the extensive deployment of IoT devices and the extent to which these devices are secured [8]. The majority of IoT devices operate on minimal power – but the cost is that they only have limited computational capability [6, 8]; they do not have the ability to run the encryption algorithms [7, 9–12] traditionally used on desktop and laptop computers. Such limitations have emboldened malicious actors to target these types of devices in particular [6].

To address this problem, researchers have been developing "lightweight" encryption algorithms [8–13] that facilitate secure communications for these devices while simultaneously minimizing the amount of computation needed for encryp-

tion. In order to find the most suitable algorithm for this purpose, NIST began hosting a competition in 2019 where they encouraged the academic community to submit novel lightweight encryption algorithms [8, 10, 14]. The winner of the competition was the ASCON cipher suite; NIST announced its endorsement of ASCON in February 2023 [14]. The IEEE Standards Committee is updating 802.15.4 such that the protocol has the ability to be secured by ASCON-128a and ASCON-128 [15–17]. The IEEE standards proposal to add these two ASCON encryption algorithms: P802.15.4ae, is expected to be submitted for revisions in December 2026 [15, 17].

Much previous work has been done to examine the performance and security of ASCON [14, 18–23], but there has been little research [21, 23] that examines ASCON in real world environments. There was no research that tested the use of ASCON on commercial smart home devices. This thesis examines the potential of the ASCON cipher suite in securing the communications of smart home devices when using the Thread [24, 25] commercial IoT protocol, and how OpenThread [26], Google's open source implementation of Thread, can be modified to use ASCON encryption. A comparative study is conducted to show that when OpenThread is secured with ASCON, the delay, throughput, packet loss, and battery life of smart home devices are not adversely affected compared to when using OpenThread unmodified, secured with AES-CCM.

This thesis is structured into five further chapters. Chapter 2 gives a detailed description of AES, ASCON, Thread, and CoAP, along with a discussion on related work, and describes the hardware platform used in this thesis. Chapter 3 describes the replacement of the existing symmetric cipher with ASCON. Chapter 4 discusses the experimental setups for comparative studies. Results of experiments are presented in Chapter 5. Chapter 6 concludes the thesis, and discusses the possibilities for further work.

Chapter 2

Background

2.1 AES

2.1.1 Overview

AES, formerly known as Rjindael, is the encryption algorithm that won the "Advanced Encryption Standard (AES)" competition hosted by NIST from 1997 to 2000 [27, 28]. NIST chose Rjindael because, relative to the competing algorithms, it was able to effectively balance performance without sacrificing the security of encrypted information. After endorsing Rjindael as the winner in 2000, NIST created three variations of Rjindael that would be available to the U.S. government and commercial organizations: AES-128, AES-192, and AES-256 [27]. In this thesis, the acronym "AES" will be used to refer to the Rjindael algorithm

and all variations of it that were made by NIST, and not the competition.

2.1.2 Encryption Algorithms

	Key length		**Block size**		**Number of rounds**
	Nk	(in bits)	*Nb*	(in bits)	*Nr*
AES-128	4	128	4	128	10
AES-192	6	192	4	128	12
AES-256	8	256	4	128	14

Figure 2.1: Differences between AES-128, AES-192, and AES-256 [27].

The AES specification [27] states that the encryption algorithms: AES-128, AES-192, and AES-256, protect the confidentiality of plaintext data by manipulating it through the use of substitution operations. The number of *rounds* of substitution operations performed on the plaintext or ciphertext, during encryption and decryption respectively, differ between the encryption algorithms. The length of the symmetric key used between the algorithms also differ in size: AES-128, AES-192, AES-256 utilize keys of length 16, 24, and 32 bytes, respectively. Under each of the three algorithms, the ciphertext is always the same length as the plaintext.

A disadvantage with AES-128, AES-192, and AES-256 is that they are unable to verify the integrity of ciphertext data during decryption. However, NIST

has created "block cipher modes" [29, 30] of AES, modified versions that provide confidentiality and integrity for the plaintext being encrypted.

2.1.3 AES-CCM

The AES-CCM encryption algorithm is a "block cipher" variation of AES that, beyond protecting the confidentiality of plaintext data, can determine the integrity of ciphertext data [30]. AES-CCM is used in the 802.15.4 standard [31,32] and in Thread [33].

AES-CCM requires the plaintext and a symmetric key of 16 bytes in length, plus a nonce of arbitrary length. The nonce and associated data are used by AES-CCM during encryption create an arbitrary length Message Authentication Code (MAC), which is used during decryption to verify the ciphertext. Similar to the original AES algorithms, the length of the plaintext and corresponding ciphertext is always the same.

2.2 ASCON

2.2.1 Overview

ASCON is a cipher suite composed of multiple encryption and hashing algorithms designed for devices with limited power and computational abilities [1,18].

The ASCON AEAD algorithms ensures the confidentiality and integrity of in-flight packets, and can withstand side-channel attacks that occur when malicious actors are able to observe the behavior of compromised hardware. The original ASCON specification [18] defines ASCON-128, ASCON-128a, and ASCON-80pq for AEAD, along ASCON-HASH and ASOCN-XOF for hashing.

ASCON was chosen as the best lightweight cipher suite in the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) in February 2019 [34], and NIST announced ASCON as the winner of its lightweight encryption competition in 2023 [14]. NIST [14] state that the ASCON AEAD and hash algorithms displayed the best results in terms of performance and security relative to other candidates. Furthermore, a number of independent studies [14, 18–23] have evaluated the security and performance of the ASCON algorithms. A group of researchers has described the modifications that they recommend NIST to make to ASCON before releasing a standardized version [35]. These recommendations include modifying the ASCON AEAD ciphers to enable the use of tags less than 16 bytes in length, and removing ASCON-80pq from the cipher suite.

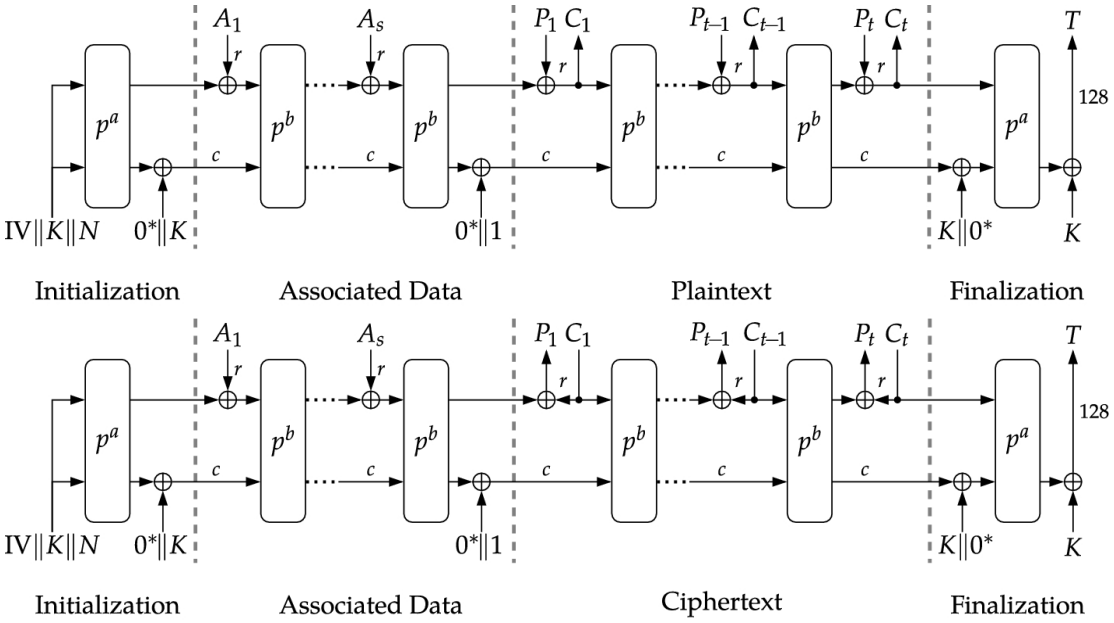


Figure 2.2: The top and bottom diagrams display the encryption and decryption processes, respectively, under ASCON AEAD [18].

2.2.2 ASCON AEAD

In the original specification, ASCON-128a, ASCON-128, and ASCON-80pq are defined for the purposes of AEAD, which all utilize repeated permutations to convert plaintext into ciphertext during the encryption process. All three ciphers take as input a key of no greater than 20 bytes, a nonce that is 16 bytes in length and associated data of arbitrary length. These inputs are used to encrypt plaintext into ciphertext; the same inputs must be used to decrypt the ciphertext back into its original form. The ciphertext is always the same size as the ciphertext. A tag of 16 bytes is always returned after encryption, and is used during decryption to verify integrity of the ciphertext.

The difference between ASCON-128a and ASCON-128, to ASCON-80pq, is that the former two variants only support a maximum key size of 16 bytes, while ASCON-80pq can support keys that are up to 20 bytes long. The inputs and outputs are identical between ASCON-128a and ASCON-128; however, they differ with respect to the number of permutations they used during encryption and decryption, and the number of bytes they can encrypt and decrypt at once, which the specification defines as the *block size*. As shown in Figure 2.3, ASCON-128a utilizes a smaller number of permutations and supports a smaller block size than ASCON-128. ASCON-80pq utilizes the same block size and number of permutations as ASCON-128.

Name	Algorithms	Bit size of				Rounds	
		Key	Nonce	Tag	Data block	p^a	p^b
Ascon-128	$\mathcal{E}, \mathcal{D}_{128,64,12,6}$	128	128	128	64	12	6
Ascon-128a	$\mathcal{E}, \mathcal{D}_{128,128,12,8}$	128	128	128	128	12	8

Figure 2.3: Differences between ASCON-128a and ASCON-128 [18] . The variables p^a and p^b represent the number of permutations performed on the associated data, and the plaintext or ciphertext, respectively.

Name	Algorithms	Bit size of				Rounds	
		Key	Nonce	Tag	Data block	p^a	p^b
Ascon-80pq	$\mathcal{E}, \mathcal{D}_{160,64,12,6}$	160	128	128	64	12	6

Figure 2.4: The values for the nonce, key, associated data, along with the block size and number of permutations used, for the ASCON-80pq cipher [18].

2.2.3 Source Code

2.2.3.1 ASCON-C

The creators of ASCON have made their C implementations of their ASCON AEAD and hashing algorithms publicly available on GitHub [4]. They designed a general "reference" ASCON C implementation, and provided implementations of each ASCON algorithm optimized for specific hardware, which include ASCON hashing and AEAD implementations written for ARM and ESP32 devices. On their official website [36], the creators of ASCON endorse hardware implementations of the ASCON algorithms. On GitHub [37], they maintain a list of endorsed software implementations of ASCON written in programming languages such as Python, Rust, and Javascript.

2.2.3.2 LibAscon

LibAscon [3] is a third party C implementation of ASCON endorsed by the creators of ASCON. LibAscon adds new features to ASCON that are not defined

in the ASCON specification [18], such as allowing for the creation of tags that can be larger or smaller than 16 bytes in the ASCON AEAD algorithms. LibAscon also provides an API to enable developers to write code that does ASCON AEAD encryption and decryption over *multiple* function calls. The ASCON-C reference implementation only defines functions that conduct ASCON AEAD over a single function call. However, a disadvantage of LibAscon is that its implementation of the ASCON algorithms are not optimized for specific hardware, and it is not as secure as the official ASCON-C implementations.

2.3 CoAP

The Constrained Application Protocol (CoAP), defined in RFC 7252 [38], is designed to enable resource-constrained devices to produce and consume RESTful APIs at the application layer. CoAP brings over the primary features of the Hypertext Transfer Protocol (HTTP) for use by resource-constrained devices. CoAP implements the same client-server model seen in HTTP, in which a client sends a request to a server, which then sends the client back a response. Commonly used status codes in HTTP — `200` to indicate success, or `404` to state when a given resource does not exist, are present in CoAP. The concept of a Universal Resource Identifier (URI) to represent the location of a given resource is included in CoAP as well.

Unlike HTTP, CoAP utilizes UDP at the transport layer. Mechanisms for reliable packet delivery are implemented in CoAP itself, and not at the transport layer. The header of all CoAP messages contain a `type` field that indicates whether reliable delivery is needed: packets that use reliable delivery are set to have the value "Confirmable", while packets in which retransmission is not needed will have the value "Non-Confirmable". Acknowledgements are used to indicate successful reception of Confirmable packets. Retransmissions are used when Confirmable packets do not initially make it to their destination.

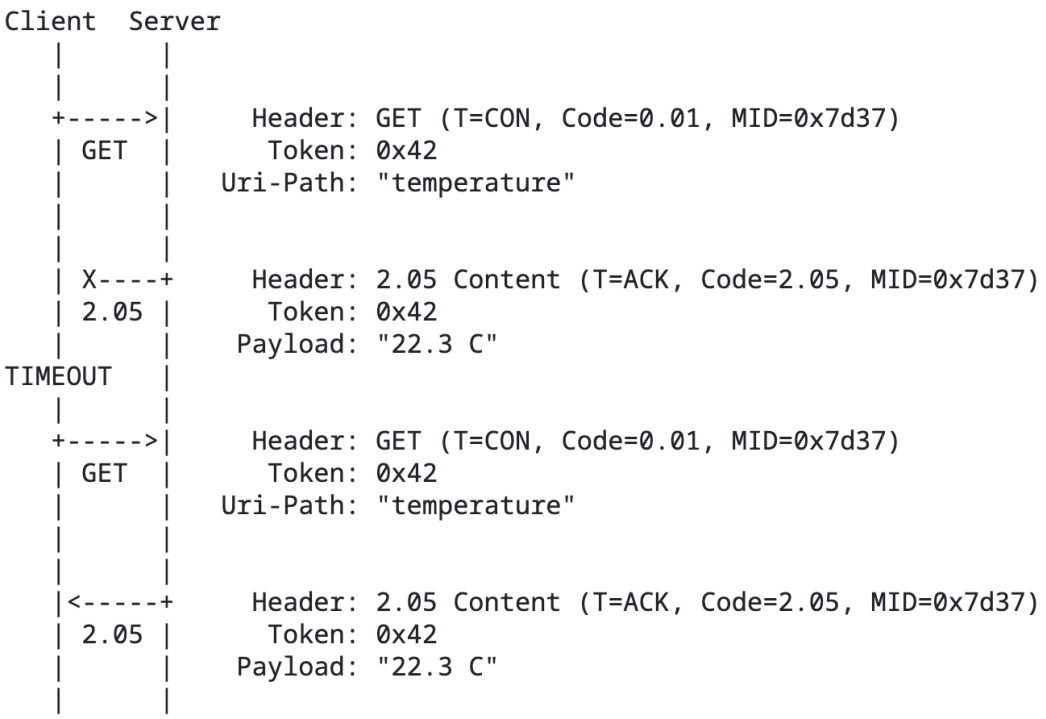


Figure 2.5: A timing diagram showing how a retransmission occurs when an Acknowledgement for a Confirmable request is not received [38].

Since CoAP is made for resource-constrained deployment scenarios, packets are designed to be as small as possible to avoid fragmentation at the network layer. CoAP clients utilize multicasting[1] to send request to multiple different servers, a feature not present in HTTP. The Datagram Transport Layer Security (DTLS), a modified version of Transport Layer Security (TLS) that operates under UDP [39], is used to provide encryption to CoAP payloads. DTLS and CoAP are used in Thread [33, 40].

RFC 7641 [41] defines an additional feature for CoAP known as the Observe protocol, where the server hosts a RESTful endpoint corresponding to a specific resource. In order to receive state updates for a resource over time, a client must *register* with the server by sending a GET request, which has the CoAP `observe` header is set to `0`. Upon receiving this request, the server sets up a subscription for the client in which the server will send CoAP responses about the state of the resource. These responses can be sent by the server at fixed intervals, or whenever there is an update to the state of the resource. The Observe feature of CoAP enables a client to receive a steady stream of responses regarding the state of particular resources it is interested in, without having to send a GET request every time whenever it needs the latest information about a given resource.

[1]However, RFC 7252 [38] states that it is not possible for clients to use multicasting when DTLS is enabled.

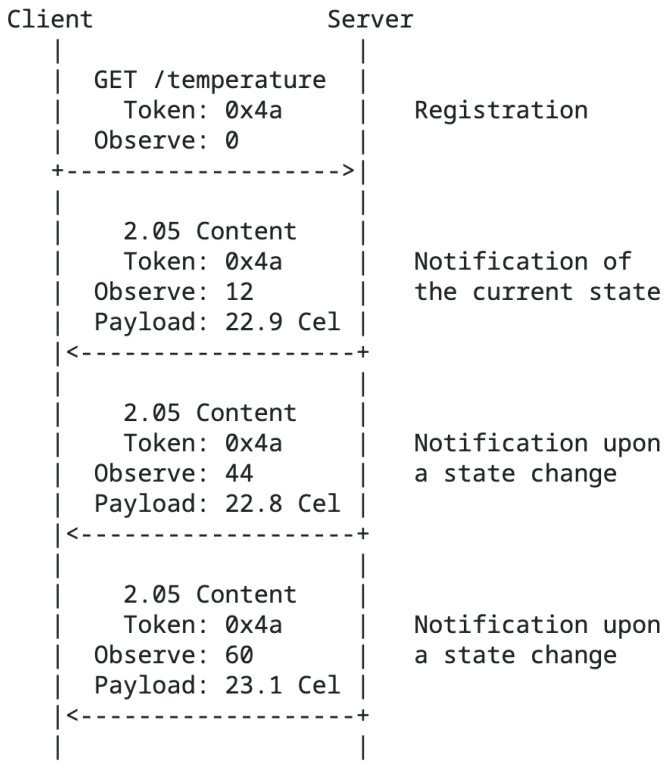


Figure 2.6: A timing diagram displaying a CoAP observe interaction between a client and a temperature sensor, which is running the CoAP server [38]. The measurement unit `Cel` corresponds to degrees Celsius.

2.4 Thread

2.4.1 Overview

Thread is a wireless mesh network protocol developed primarily for smart homes [24, 25, 42] and smart buildings [43, 44]. Thread is built on IEEE 802.15.4, a wireless MAC and physical layer standard designed for resource-constrained IoT devices [33]. This enables Thread to support battery operated devices [45, 46]

and have built-in encryption at the MAC layer. A single Thread network can contain 250 devices [42], and has been designed to be user-friendly [47]. The end user connects their Thread devices through a process known as *commissioning* [40, 46] Once commissioned, the logistics of how each device communicates with one another are handled by Thread [46].

The level of sophistication in which these devices can automatically handle their communications is a key feature of Thread [33,46]. When the "leader" device of a Thread network goes offline, other devices have the ability to collectively decide on a new leader to ensure the stability of the network [33, 46]. Groups of devices unable to reach one another are capable of forming temporary network *partitions* [33, 48] until they are able to communicate more widely and combine back into a single network. Such features of Thread allow to protocol to be described as "self-healing" [46].

Thread is a relatively new protocol rising in popularity [42]. The specification [33] is managed by the Thread Group, whose members are composed of well-known companies, including Google, Apple, Amazon, and Samsung [49]. Manufacturers and product developers must undergo a certification process before they can sell their Thread devices to customers [50].

The network stack for Thread defines the physical layer and up until the transport layer, enabling developers to utilize their own protocols at the application

layer [46, 51].

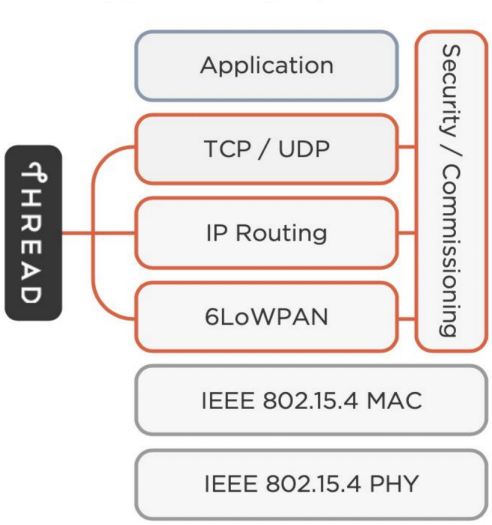


Figure 2.7: The Thread network stack [51].

Each layer in the Thread network stack takes advantage of an already established network protocol: IEEE 802.15.4, IPv6 over Wireless Personal Access Networks (6LoWPAN), and UDP. The use of these protocols allows Thread to be robust and secure without needing to develop novel solutions. In Thread, all devices natively use Internet Protocol Version 6 (IPv6) addressing to uniquely refer to one another [46, 51].

2.4.2 Types of Devices & Network Topology

The Thread Group [33, 46, 48] state that devices in a Thread network are classified as follows.

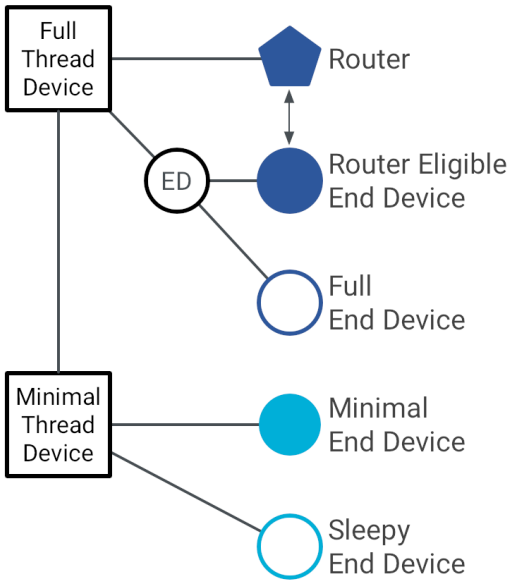


Figure 2.8: Device types in a Thread network from the OpenThread documentation [48].

- **Full Thread Device (FTD)**: a device powered on at all times, with the ability directly communicate with "neighboring routers" [33, p. 2-9]. The different types of FTDs are:
  - **Router**: responsible for forwarding packets between a source and destination.
  - **Router Eligible End Device (REED)**: has the capability to be a router, but does not currently need to be one[2]. A FTD can become a router at any moment when one is needed in the network. For example, a REED becomes a router when there is a Thread device needs to

[2]According to [48], a Thread network can only support a maximum of 32 routers. When a network has $n > 32$ FTDs, the $n - 32$ FTDs will ending up being REEDs.

communicate with a router, but there are no nearby routers for the device to communicate with, and the REED is the device nearest to it [33, 46].

  - **Full End Device (FED)**: a FTD *not* capable of performing router responsibilities.

  - **Border Router**: a FTD that primarily provides the ability for devices to communicate *outside* the Thread network. Thread devices send their packets to the Border Router, which is able to send those packets to devices outside the Thread network. All FTDs have the ability to become a border router.

- **Minimal Thread Device (MTD)**: is incapable of routing packets, and is not necessarily powered on all the time. MTDs are only capable having a direct connection to their parent router, the router that receives and sends packets on their behalf. Some MTDs have the ability to go to sleep when inactive.

  - **Minimal End Device (MED)**: a MTD that is powered on at all times.

  - **SED**: a MTD that has the ability to go to sleep. SEDs wake up at predefined intervals to communicate with a router to send and receive

packets from other devices in the Thread network.

- **Synchronized Sleepy End Device (SSED)**: a variation of the SED which, rather than sending packets to its parent router check for received packets, it will instead "listen" [46] and wait for its parent to forward packets the parent has received on behalf of the SSED. How often this exchange occurs is defined by a predefined interval agreed upon by the parent and SSED. A SSED and their parent interact by using Coordinated Sampled Listening (CSL) as defined in 802.15.4 [33, 46].

Routers make up the mesh topology of any Thread network [46]. However, this does not imply that Thread networks are true mesh networks in the sense that every device has a direct connection with one another. Since MTDs can only directly communicate with their parent routers, routers with children will form small "hub and spoke" connections at the edge of the mesh network. Routers form the backbone of the mesh network, while the non-router FTDs and MTDs form "hub and spoke" clusters with their parent routers at the edge of the mesh network. Border routers facilitate the communications between the phone (connected to Wi-Fi) and the Thread devices.

2.4.3 OpenThread

OpenThread [26] is an open source implementation of Thread created by Google, written in C/C++. The source code is endorsed by commercial organizations such as Amazon, NXP, and Silicon Labs [52]. A plethora of hardware companies, including Espressif, Nordic Semiconductors, and Silicon Labs, have developed 802.15.4 microcontrollers that can natively run OpenThread [53]. Google provides guides [53] to show developers how Thread works, and how to use OpenThread to write application layer software.

2.4.4 Thread in Smart Homes

The smart home deployment scenario is a primary use case for Thread [24, 25, 42]. The ability for Thread to support battery powered devices makes it well-suited for smart homes sensing devices which operate on battery power. Border router functionality can be given to smart home devices that already serve a different purpose, such as an Amazon Echo [54] or a mains power smart light. The Thread Group notes that negates the need for a separate device operating solely as a border router.

The smart home use case is such an essential part of Thread that the protocol plays a major part in a smart home protocol known as Matter [24, 25, 42], also known as Connected Home Over IP (CHIP) [55, 56]. Matter is an application

layer protocol which enables smart home devices to easily communicate with each other via Wi-Fi, Thread, Zigbee[3], and Bluetooth Low Energy (BLE) [57–59].

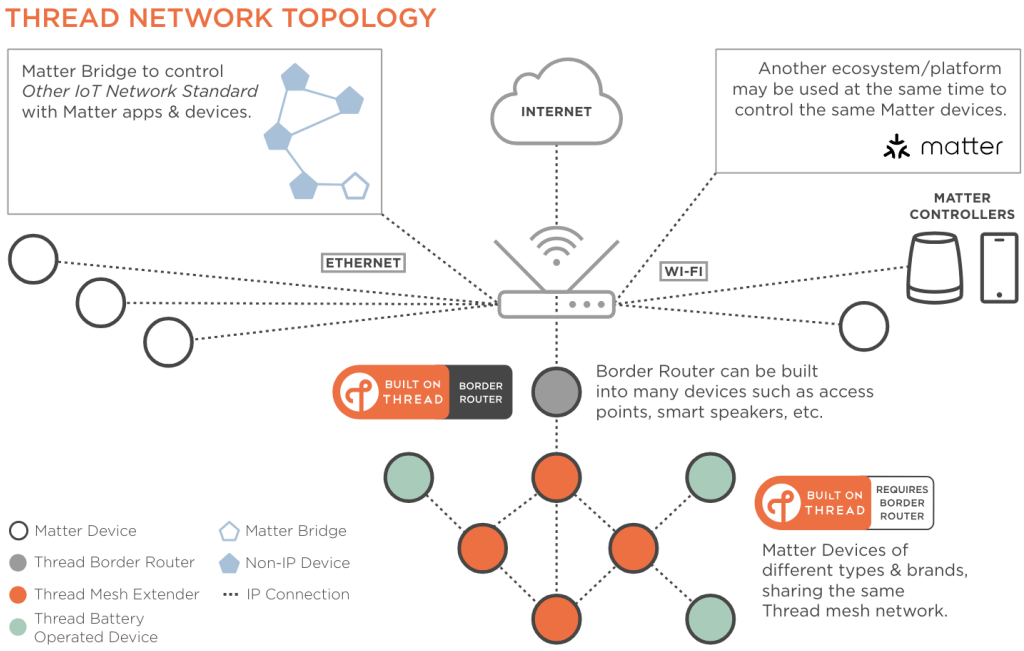


Figure 2.9: The topology of a typical smart home network under Thread [44]. A Thread smart home network has the ability to communicate to non-Thread devices that support the Matter application layer protocol.

[3]Matter does not support Zigbee by default; a *bridge* must be used to connect Zigbee devices to a Matter network [57, 58].

2.5 ESP-IDF

2.5.1 Overview

ESP-IDF is a software development platform from Espressif, which give developers the ability to write C programs for ESP32 MCUs [60, 61]. Espressif is a member of the Thread Group [49] and endorses Google's OpenThread source code [52]. The OpenThread source code is included in ESP-IDF [62], enabling developers to program their 802.15.4 ESP32 microcontrollers to operate under the Thread protocol. Espressif also offers an SDK allowing developers to use Matter in their smart home ESP32 devices [59]. The ESP-IDF codebase is open source and is available on GitHub [60].

2.5.2 ESP32-H2

The ESP32-H2 is an Espressif SoC with the ability to run BLE and 802.15.4 [65]. As a result, the ESP32-H2 is capable of supporting Zigbee and Thread. However, unlike the ESP32-C6, an Espressif SoC that can utilize 802.15.4 and Wi-Fi [66], the ESP32-H2 is incapable of operating under Wi-Fi. As a result, the ESP32-H2 cannot be used as a Thread border router. The ESP32-H2 utilizes a RISC-V architecture and has the ability to perform AES encryption in hardware.

The ESP32-H2 SoC can be purchased as a chip or module that can be used

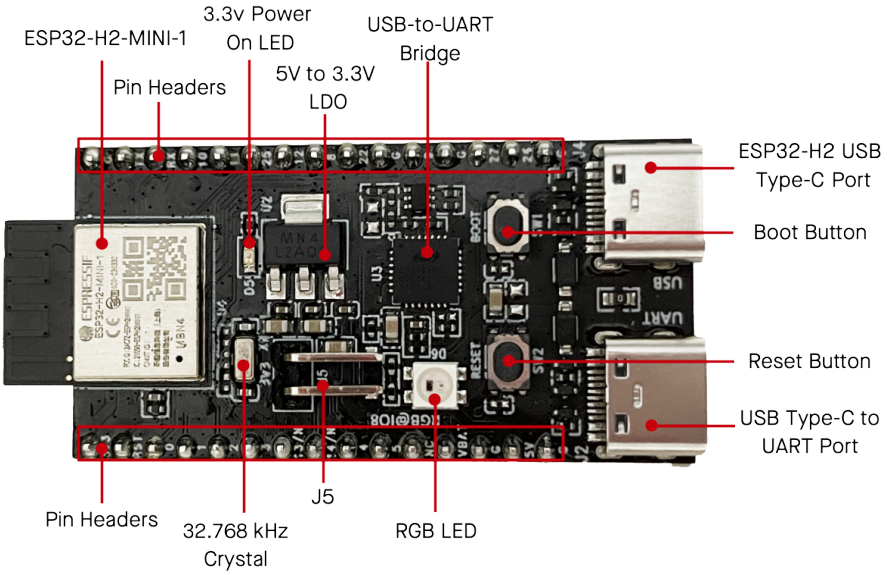


Figure 2.10: The ESP32-H2 development kit [63]. The component labelled "ESP32-H2-MINI-1" is the SoC that can be attached as a part of a larger MCU [64].

as a component for a larger MCU [64, 67]. Espressif also sells the ESP32-H2 as a development kit (Figure 2.10) in which the SoC is soldered onto a MCU, enabling developers to program the ESP32-H2 without having to worry about the hardware setup [63].

2.5.3 ESP Thread Border Router

Espressif sells prebuilt boards capable of operating as Thread border routers[4] [68, 69]. The ESP Thread Border router board utilizes two SoCs: the ESP32-H2 and ESP32-S3, capable of using Thread and Wi-Fi, respectively. Developers can write programs for their Thread border routers using the Espressif Thread

Border Router SDK [69]. The ESP Thread Border Router utilizes the OpenThread Radio Co-Processor (RCP) design: where the attached ESP32-H2 SoC manages the 802.15.4 MAC and Physical layers, while the ESP32-S3 SoC runs the network, transport, and application layers [62, 68–70]. The ESP32-H2 and ESP32-C6 SoCs [70, 71] communicate using the Spinel protocol [72].

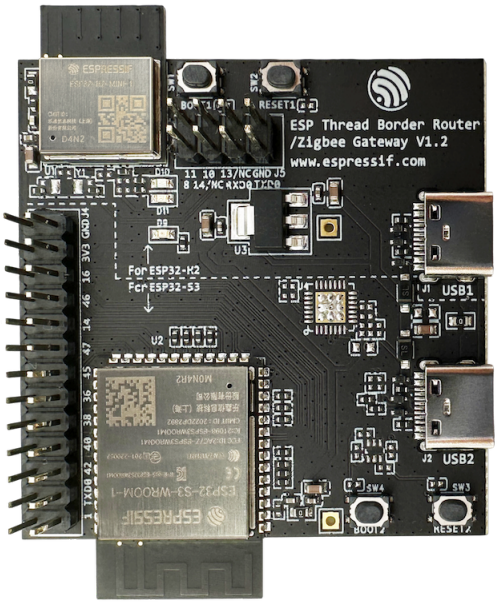


Figure 2.11: Prebuilt ESP Thread Border [68, 69]. The top and bottom modules are the ESP32-H2 and ESP32-S3 SoCs, respectively.

[4]The device can also function as a Zigbee gateway [68].

2.5.4 ESP32 Deep Sleep

The ESP-IDF platform provides developers the ability to choose whether to program their ESP32 Thread SEDs to operate under deep sleep [73] or light sleep [74]. Espressif [74] state that Thread sleepy devices under *light sleep* will function identically as SEDs operating to the Thread specification: their transceiver is off when they are not sending packets, and they will poll their parent router at predefined intervals to obtain packets it has received [33, 75]. However, SEDs operating under *deep sleep* function differently: the majority of the device, including the transeciver and the CPU, are off during deep sleep [75]. A SED under deep sleep must "reboot" [73] on wakeup and reconnect to the network through the Thread network attachment process [33].

Espressif's definition of deep sleep is different from the Thread Group's use of the term. In the Thread "Battery Operated Devices" whitepaper [45], the term is implicitly used to describe the scenario when the SED's transceiver is off. Espressif states that their definition of *light sleep* matches how the Thread Group defines the behavior of SEDs in [33, 45]. It is Espressif who makes the explicit distinction between light sleep versus deep sleep.

2.6 Related Work

2.6.1 ASCON in Real World Deployment Scenarios

The literature has focused on examining ASCON within a controlled environment [14, 18–23]. There has been relatively little research exploring how ASCON can be used in realistic deployment scenarios, yet it has shown the potential of ASCON in such use cases. Leading up to their Feburary 2023 announcement [14], researchers at NIST [21] examined the performance of ASCON when securing MQTT packets in a weather monitoring scenario. They found that use of ASCON AEAD did not adversely impact the RAM and CPU performance of devices in MQTT, compared to when packets were not encrypted. Researchers at Texas State University [23] compared the performance of AES and ASCON when securing packets sent by environmental sensors operating under the Long Range (LoRa) protocol. The researchers discovered that ASCON utilizes less memory and power than AES, and that the difference in power consumption, measured in Watts, between ASCON and AES increased when devices were communicating over longer distances.

Previous work has investigated how ASCON operates as part of a larger network. Researchers have used ASCON-128a in a 6LoWPAN key exchange algorithm [76] — and showed that their algorithm consumed less energy on resourced

constrained devices, compared to when the devices were using other 6LoWPAN key exchange algorithms. More recently, a group of researchers has utilized ASCON AEAD to create a system that allows smart homeowners to protect the confidentiality of sensitive digital information such that it can only be accessed when the users are at home [77].

2.6.2 Performance of Thread

The performance of Thread has been examined in smart buildings and offices [78–82], as had the energy consumption of SEDs operating in an office environment; a model has been produced to predict the energy consumption of SEDs in various scenarios [83, 84]. However, there has been no examination of the network performance of SEDs when running on *smart homes* devices. No research that has examined how ASCON AEAD can be used to secure Thread, and whether doing so is feasible.

Chapter 3

Replacing the OpenThread Encryption Algorithm

This chapter describes the modifications made to OpenThread such that the ASCON-128a and ASCON-128 AEAD encryption algorithms are used in place of AES-CCM. Included is a description of the challenges encountered and how they were solved, along with a discussion of the outstanding challenges for future work.

3.1 Overview

The primary goal is to modify OpenThread to secure the communications of ESP32 Thread devices with ASCON AEAD. Since OpenThread is hardware-agnostic, more work was needed beyond making changes to the official GitHub

repository [52]. Modifications had to be made in the ESP-IDF source code, which includes Espressif's fork of OpenThread as a Git submodule [85]. ESP-IDF provides the Espressif OpenThread fork as a dependency that can be imported into a ESP-IDF project whenever developers need to run OpenThread on ESP32 hardware. The source codes of OpenThread and ESP-IDF are implemented in `C++` and `C`, respectively.

The two repositories forked were the ESP-IDF codebase and Espressif's OpenThread fork (rather than the official OpenThread repository). My forks of ESP-IDF and the Espressif OpenThread fork can be found in the `UCSC-ThreadAscon` GitHub organization [86]. Commits are frequently being made[1] to the ESP-IDF source code and the Espressif OpenThread fork. The forks are updated to be up-to-date with the `main` or `master` branches of their respective upstream repositories whenever new changes are made.

The forks of ESP-IDF and the Espressif OpenThread fork collectively make up my modified version of OpenThread which run the LibAscon implementations of ASCON-128a and ASCON-128 on 802.15.4 compatible ESP32 hardware. This modified OpenThread implementation enables users to specify whether ASCON-128a or ASCON-128 should be used for encryption with the use of a `Kconfig`

[1]Changes to the Espressif OpenThread fork [85] are made on a near daily basis, while the ESP-IDF repository is updated usually once or twice a month, especially when there is a new release. Espressif always keeps its OpenThread fork up to date with the `main` branch of Google's OpenThread repository.

variable. The forks do not support ASCON-80pq, as the creators of ASCON encourage using ASCON-128 and ASCON-128a instead [18]. Before NIST's released its official version of ASCON, a recommendation was made to remove ASCON-80pq from the cipher suite [35].

3.1.1 The nRF Packet Sniffer

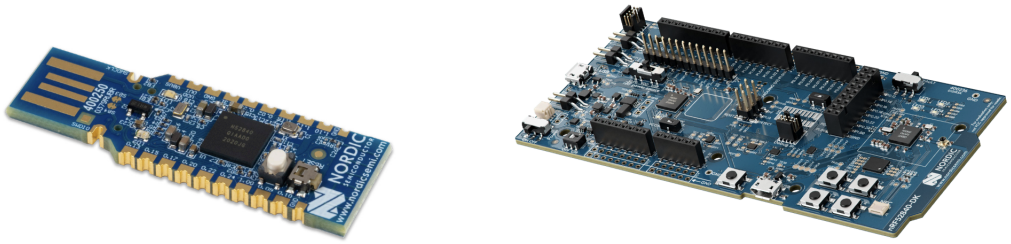


Figure 3.1: The nRF52840 dongle (left) [87] and the nRF52840 development kit (right) [88].

A useful tool used was the nRF 802.15.4 packet sniffer software by Nordic Semiconductors [89], which is designed to be run on either the nRF52840 development kit or the nRF52840 dongle. The nRF52840 dongle was set up to run the nRF 802.15.4 packet sniffer to listen to and display the packets sent between the ESP32 devices on Wireshark [90]. Since Thread is encrypted by AES-CCM, Wireshark was given the Thread network key used by the ESP32 devices so that it could properly decrypt and parse the contents of in-flight packets [91].

3.2 Removing AES-CCM

Before ASCON-128a and ASCON-128 could be added to OpenThread, AES-CCM must first be removed. Once the ESP32 devices can communicate in plaintext, ASCON encryption could be added. AES-CCM is used in Thread [33] during the encryption and decryption of payloads for:

1. Management packets send by the Mesh Link Establishment (MLE) protocol, which occurs on top of UDP at the transport layer.

2. Data frames (i.e. non-MLE packets) at the MAC layer.

The encryption and decryption of payloads are handled separately for MAC frames and MLE packets. The payload of MLE packets are encrypted in software. The payloads of frames carrying application layer data are encrypted in hardware, while frames sent during *commissioning*, the process by which a new device joins a Thread network [33], are encrypted in software. All payloads are decrypted *in software*, regardless of packet or frame type. OpenThread implements software encryption and decryption, while ESP-IDF handles the invocation of hardware encryption.

OpenThread implements MLE encryption and decryption in a single method that is a part of the `C++` class which implements the MLE protocol. The encryption and decryption of MLE packets are only done in software. To remove AES-CCM

encryption for MLE payload, every call to the MLE encryption/decryption method in the source code needed to be comment out. This was straightforward, as this method is only called in within the class which implements MLE.

Due to the hardware-agnostic nature of OpenThread, the source code itself performs software encryption on MAC frames carrying application layer data. Immediately before software encryption would begin, the Espressif OpenThread fork redirects the encryption to the hardware, preventing execution of the built-in software encryption of OpenThread. The line of code hardware encryption is located in ESP-IDF. I commented out this line to remove the hardware encryption of application layer data.

Software encryption and decryption are defined as separate methods for a `C++` class[2] which represents a MAC frame. To remove the software encryption and decryption of MAC frames, all invocations of these class methods present throughout the OpenThread source code needed to be removed. This was straightforward, as all calls were located in source files which implement the 802.15.4 MAC layer of OpenThread.

[2]The methods for software encryption and decryption are defined in two separate `C++` classes. The method which implements software encryption is defined as part of the `Frame` class, which represent MAC frames to be transmitted, while the software decryption method is implemented for `RxFrame`, representing frames received by a Thread device. The two classes are subclasses for the `Frame` class, which is the base class for a 802.15.4 MAC frame.

3.2.1 Confirming Removal of AES-CCM

The following approach was taken when removing AES-CCM:

1. Remove hardware and software encryption for MAC frames, while keeping AES-CCM MAC decryption (along with MLE encryption and decryption) intact. Use Wireshark to confirm that the payload of MAC frames are in plaintext. The ESP32 devices should not be able to receive frames with plaintext payloads.

2. Remove MAC decryption. The ESP32 devices should now be able to communicate when the frames are sent in plaintext. MLE packets should still be encrypted, as the removal of MLE encryption has not yet been done.

3. Remove MLE software encryption. Use Wireshark to confirm that MLE payloads are in plaintext. The ESP32 devices should be able to receive plaintext MLE packets.

4. Remove MLE software decryption. At this stage, the ESP32 devices should now be able to communicate completely in plaintext.

This approach was adopted to have a means of confirming the success of each incremental removal of AES-CCM. Wireshark attempts to decrypt packets using AES-CCM. It cannot properly parse most[3] plaintext packets, as it attempts to

decrypt them. The inability of Wireshark to parse plaintext is used to determine whether encryption has been successfully removed at each step.

For frames carrying application layer data, I was able to confirm that packets were sent in plaintext by instructing an ESP32-H2 device to send a UDP packet containing human-readable text as payload, and see that human-readable text in Wireshark. However, this could not be done for frames sent during commissioning and MLE packets, as the payloads of these frames and packets do not carry human-readable text. In these cases, the inability of Wireshark to successfully parse the payloads for these packets was used to confirm the removal of AES-CCM encryption.

[3]Wireshark will not decrypt certain type of packets whose data are expected to be completely in plaintext. Such packets are MLE Discovery Request and Response packets [33]. Discovery Requests and Responses are used by devices to find nearby Thread networks that they can join [46].

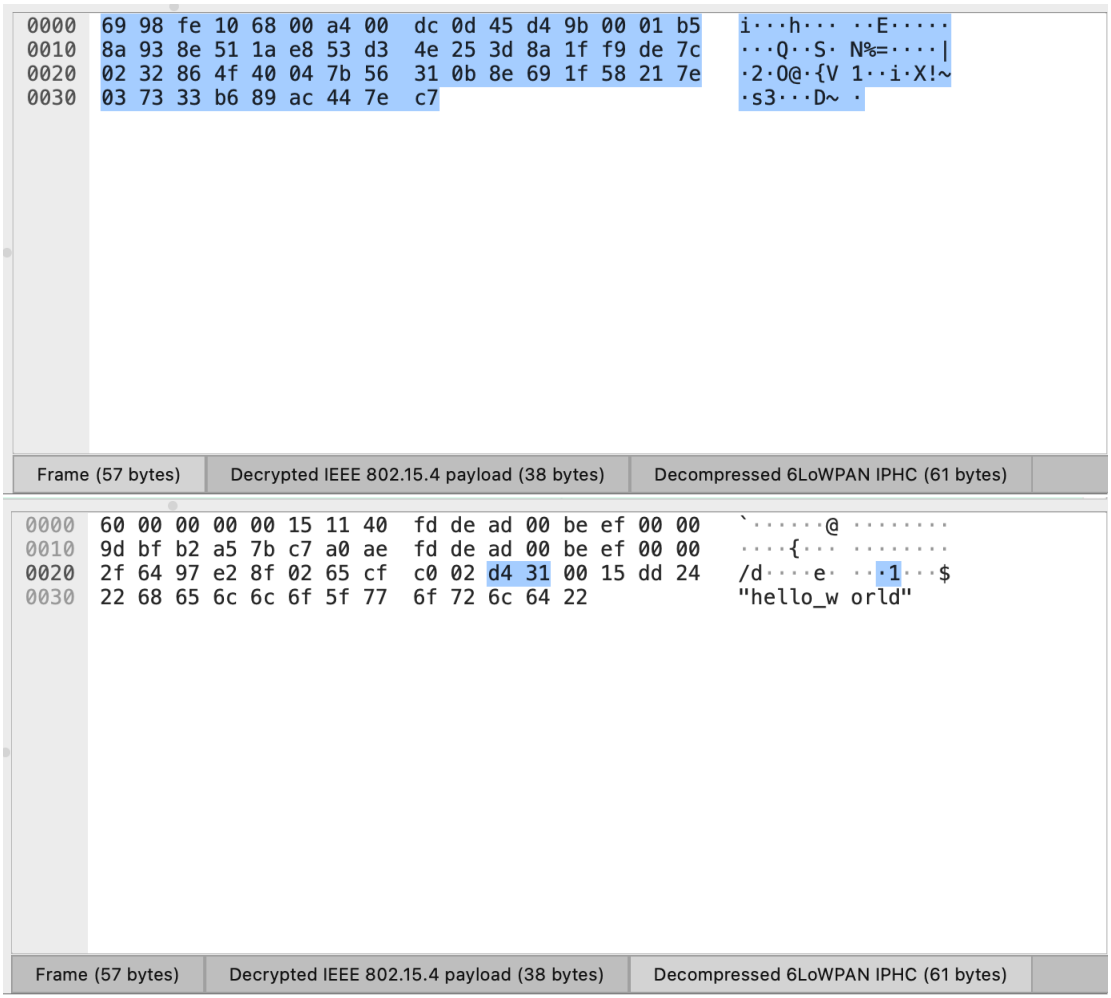


Figure 3.2: The top figures display a UDP packet that my ESP32 sent with the payload: "`hello_world`", as ciphertext encrypted by AES-CCM. The bottom figure shows payload after Wireshark has decrypted it.

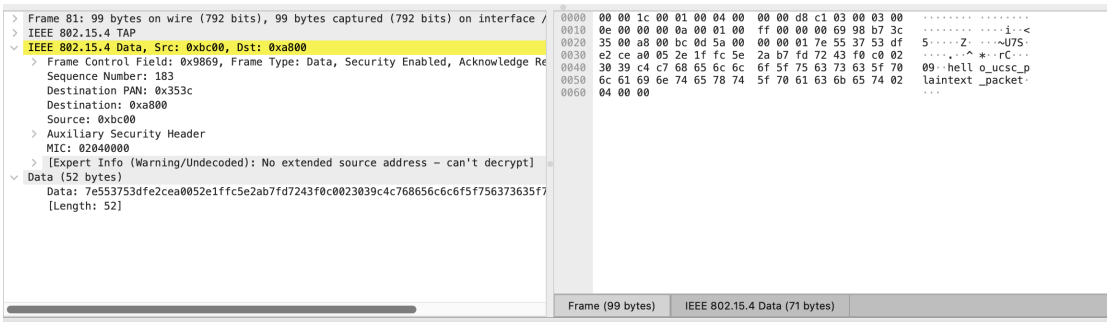


Figure 3.3: A UDP packet with the payload "`hello_ucsc_plaintext_packet`" in plaintext. Note that Wireshark (unsuccessfully) decrypts this payload using AES-CCM, since all frames carrying application layer data will be encrypted in Thread.

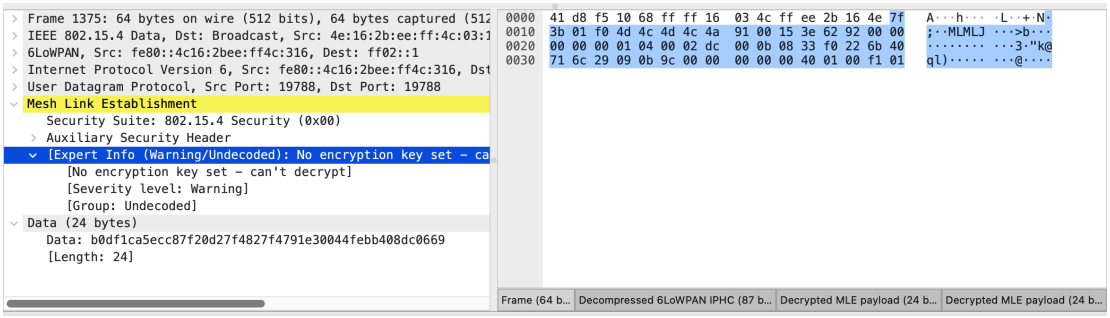


Figure 3.4: Wireshark's inability to decrypt the MLE packets proved that the removal of MLE encryption was successful.

3.3 Adding ASCON AEAD into OpenThread

For each place in which AES-CCM encryption and decryption was removed, the respective ASCON AEAD encryption and decryption functions were added. This section describes how the LibAscon implementations of ASCON AEAD were added to secure the now plaintext communications of this modified version of OpenThread.

Although there are official `C` implementations for ASCON AEAD [4], I chose instead to use the LibAscon AEAD implementations [3]. Unlike the official ASCON `C` implementations, the LibAscon implementations support the use of variable tag length. This particular feature of LibAscon enabled the creation of ASCON ciphertext with tags which are 4 bytes in length, the default size of MICs for MAC and MLE ciphertexts [33].

3.3.1 LibAscon Wrapper Functions

To enable users to switch between the use of the LibAscon implementations for ASCON-128a and ASCON-128, a `Kconfig` compiler flag was added to specify which encryption algorithm that OpenThread should use at build time. I created encryption and decryption wrapper functions that determines which respective LibAscon AEAD algorithm to use, depending on the value of the `Kconfig` variable. The modified OpenThread source code uses these wrapper functions to perform

ASCON encryption, instead of calling the LibAscon AEAD functions directly.

3.3.2 The Encryption and Decryption Process

Unlike AES-CCM, which only requires a symmetric key and nonce [30] to encrypt and decrypt ciphertexts, ASCON AEAD requires a symmetric key, nonce, and associated data [18]:

- Two separate Thread network keys are used in AES-CCM for MAC frames and MLE packets [33]. These two MAC and MLE symmetric keys are used by the LibAscon AEAD algorithms for MAC and MLE encryption and decryption, respectively. This was possible as Thread network keys are 16 bytes long [33], which is the key length required by ASCON-128a and ASCON-128 [18].

- The nonce generated for each ASCON ciphertext follows the same nonce format used by Thread with AES-CCM [33]: by using the format specified in 802.15.4-2006 [31], which is a byte sequence containing the MAC address of the sender, the frame counter, and security level field, each of which can be found in the frame header.

- Since the associated data field is not used in AES-CCM, I needed to create a custom format for the associated data. The format[4] created for the associ-

[4]The associated data used in LibAscon MAC encryption was a byte sequence of the desti-

ated data was a byte sequence of the source and destination 802.15.4 MAC addresses[5] that were present in the frame header. This ensured that every ciphertext maps to the specific "context" [92,93] of a specific source and destination device, represented by their 802.15.4 MAC addresses. This should theoretically prevent an attacker from sending ciphertext packets that which they themselves did not encrypt, or to a different destination than the one specified during the encryption of the plaintext.

Octets: 8	**4**	**1**
Source address	Frame counter	Security level

Figure 3.5: The format of a nonce as input to AES-CCM encryption, as described in 802.15.4-2006 [31].

Four functions were created to support LibAscon AEAD: two functions for MAC frame encryption and decryption, and two functions for doing the same operations on MLE packets. These MAC and MLE encryption and decryption functions were defined as methods for the `C++` classes which represents a MAC

nation MAC address, followed by the MAC address of the sender. For MLE encryption, the byte sequence was reversed: the MAC address of the sender, followed by the destination. The non-consistent ordering occurred as I forgot that I placed the destination MAC address at the beginning of the byte sequence when implementing the associated data for MLE encryption. The difference in byte sequence did not have any significant impact.

[5]I developed this associated data format after learning about the purpose of associated data fields in Stack Exchange answers [92,93], which stated that the associated data property is used to match the conditions in which the ciphertext was created. Since the header of every frame contains source and destination MAC addresses, I made the associated data of each frame the concatenation of the source and destination 802.15.4 addresses.

frame and implements MLE, respectively. All four functions follow the approach described in the above bullet points in creating the nonce and associated data[6], and in obtaining the symmetric key to use in ASCON AEAD. The MAC and MLE encryption functions passed the generated nonce and associated data, the payload to encrypt, along with the respective MAC or MLE network key, to the LibAscon encryption wrapper function, which is instructed to generate a 4 byte tag. The encryption of the payload is done in-place. The tag is appended at the end of the frame header, in place of where the AES-CCM MIC would have been [31, 33]. The packet or frame containing the ciphertext payload is then forwarded for transmission.

Under MAC and MLE decryption, the nonce, associated data, ciphertext payload, the respective network key, along with the tag, is passed into the LibAscon decryption wrapper function. The payload is decrypted in place and the wrapper function returns the status code that was returned by the LibAscon decryption function, which indicates whether the encrypted payload was valid ASCON ciphertext. If the encrypted payload was valid ASCON ciphertext, the plaintext gets forwarded for further operation. Otherwise, the frame or packet is discarded.

[6]MLE associated data always use 8 byte long extended 802.15.4 MAC addresses. MAC associated data typically contains the extended MAC address when possible, but the 2 byte shortened MAC address is used instead when the extended address is unavailable, as it was not always possible to obtain the extended addresses during MAC encryption. I was able to obtain the extended addresses during MLE encryption as the OpenThread network layer provided a mechanism which enabled me to obtain an Extended MAC address corresponding to the IPv6 address of the sender or receiver (which was always available in the 6LoWPAN header). This mechanism was not available at the MAC layer.

3.3.3 Invoking LibAscon AEAD

The addition of ASCON AEAD for the parts which were previously implemented using AES-CCM in software: the encryption and decryption of MLE packets, the decryption of MAC frames, and the encryption frames sent during commissioning, was straightforward:

1. To add ASCON AEAD for MLE packets, the respective encryption and decryption functions which called the LibAscon wrapper functions were added in place of the calls to the single AES-CCM MLE encryption/decryption function that have been commented out.

2. To implement ASCON decryption for MAC frames, the class method responsible for AES-CCM MAC frame decryption was modified to call the decryption function responsible for invoking the LibAscon decryption wrapper function. After the decryption function has returned, the AES-CCM MAC decryption method returned the status code that was returned by the LibAscon decryption function.

3. To implement ASCON encryption for MAC frames sent during commissioning, the AES-CCM MAC frame encryption class method was modified to call the encryption function that invoked the LibAscon encryption wrapper function. The AES-CCM MAC encryption method returns after the

LibAscon encryption function has returned.

An additional step needed to be taken to use ASCON AEAD to encrypt MAC frames carrying application layer data, as such frames were previously encrypted in hardware by AES-CCM. A line of code precedes the call on the AES-CCM MAC frame encryption method responsible for encrypting the payloads of such frames. This line prevents the software encryption of these frames, so that they can be encrypted in hardware by AES-CCM. That line needed to be removed to enable the software encryption of application layer payloads by ASCON AEAD. After the line was removed, MAC frames carrying application layer data were now encrypted by ASCON AEAD, since the AES-CCM MAC frame software encryption method was already modified to perform ASCON encryption, as described in step 3.

3.4 Challenges & Future Work

3.4.1 A 16 Byte Tag is Too Big

The original plan was to use the official ASCON-C implementations [4]. However, since the official implementations generates tags that are 16 bytes in length, they would cause the ESP32 devices to crash if the payload carried by a MAC frame is too large. As a result, I chose to use LibAscon instead, which enable the use of 4 byte tags, the default size of the MIC used in Thread [33]. Researchers have even recommended for the ASCON AEAD algorithms to use 4 bytes, rather 16 bytes [35].

The version of the source code which uses ASCON-C can be found in my fork of the Espressif of OpenThread fork [94]. A `Kconfig` variable has been included the ESP-IDF fork [95] to run OpenThread using the official ASCON-C ESP32 optimized and reference implementations of ASCON-128a, to enable such errors to be reproduced on demand.

3.4.2 Commissioning Issue

When initially implementing LibAscon AEAD to be used for MAC layer encryption, I did not follow the nonce format described 802.15.4-2006 [31] (shown in Figure 3.5). I instead chose to make the nonce a byte sequence of the se-

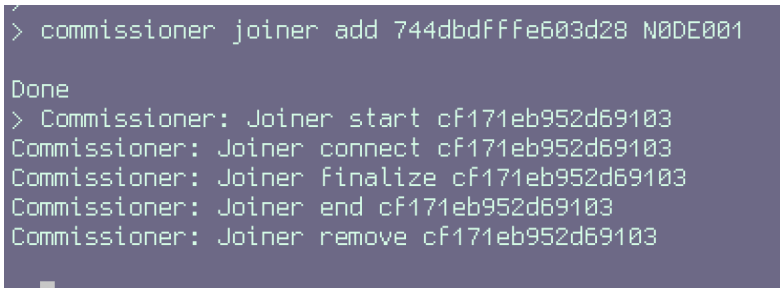


Figure 3.6: The serial monitor output of the commissioner. The commissioner successfully adds the joiner, but eventually removes the joiner after failing to communicate with it.

quence number, key ID, and frame counter, each of which were present in the frame header. The ESP32 devices were able to communicate using the LibAscon implementations of ASCON-128a and ASCON-128 with this nonce format, *except* during the commissioning process; the joiner would never be able to attach to the Thread network that the commissioner is in.

During the commissioning process, I observed that a frame that is sent by the joiner would have a *different* nonce than the nonce that is created by the commissioner upon receiving the frame. To solve this issue, all nonces needed to follow the format specified in 802.15.4-2006 [31] and that is shown in Figure 3.5. After doing so, the devices were able to successfully attach to Thread networks through the commissioning process. The 802.15.4-2006 nonce format was used not only for MAC encryption, but for MLE encryption as well, as this nonce format is used for all ciphertexts send in Thread [33].

3.4.3 Outstanding Challenge for Future Work

3.4.3.1 ASCON Encryption for SSEDs

AES-CCM hardware encryption is performed on frames carrying application layer data. However, there exists another scenario in which hardware encryption is used: when encrypting the payload of a MAC frame known as a Enhanced Acknowledgment (Enh-ACK), a special type of acknowledgement packet that was first introduced in 802.15.4-2015 [32, 96]. The payloads of Enh-ACKs are always encrypted, and are primarily used as a part of CSL, a 802.15.4-2015 feature that is implemented by SSEDs [33].

To ensure that no AES-CCM hardware encryption is done, the line of code which invokes hardware encryption for Enh-ACKs has been commented out in my fork of ESP-IDF. However, the focus of this thesis is on securing the communications of the primary Thread device types: FTDs, MTDs, SEDs, and Thread Border Routers. Since a SSED is a special type of SED, securing the communications of SSEDs using LibAscon AEAD ciphers is not within scope. Future work can explore modifications to make in order to secure SSEDs using ASCON AEAD, and the impact of network performance and energy consumption of SSEDs when ASCON AEAD is used over AES-CCM.

Chapter 4

Evaluation

To determine the effectiveness of Thread when using ASCON, experiments were conducted to examine the energy consumption and network performance of ESP32 Thread devices when running the modified version of OpenThread that uses LibAscon encryption. The experiments were conducted in a simulated smart home environment. Another experiment was conducted to examine ciphertext payloads sent by the ESP32 devices to confirm that the modified version of OpenThread properly uses ASCON. The driver code for all experiments are available on GitHub [97–100].

4.1 Correctness Tests

The LibAscon decryption implementations for ASCON AEAD return a status code to indicate if a sequence of bytes ciphertext is valid ASCON. If the status codes returned by the decryption function indicates that all ciphertexts are valid, then the modified version of OpenThread properly utilizes LibAscon AEAD. These experiments only test the devices' ability to encrypt using LibAscon; if the devices can encrypt LibAscon ciphertexts, then the devices can decrypt LibAscon ciphertext as well, since the devices were already able to communicate with each other when running the modified implementation of OpenThread.

A Thread network was set up that contained two ESP32 Thread devices. A computer (my MacBook) was used to monitor the command line outputs of both devices. The devices were instructed to hex dump the ciphertext payload of packets that were sent, or which has been received before decryption, along with the nonce, associated data, symmetric key, and tag corresponding to the ciphertext. A small subset of the hex dumped data was selected as samples to be passed to the separate computer to be decrypted using LibAscon. The `C` programs and the sample ciphertexts used in the experiments are available on GitHub [101].

4.1.1 Limitations

The creators of ASCON maintain a list of software implementations of the cipher suite created by themselves and outside contributors [37]. To ideally test whether the modified implementation of OpenThread is properly secured by ASCON AEAD, a separate computer should use a *different* implementation of ASCON, rather than LibAscon. This was not possible, as the Correctness Tests require an implementation of ASCON-128a and ASCON-128 that allow for variable tag length for the reasons discussed in Section 3.4.1.

, all of which are "older" AEAD variants that were *not* defined in the original pre-NIST specification of ASCON [18]. When this thesis was written, `pyascon` did not have an implementation for ASCON-128a.

4.2 Delay Tests

The Delay experiments measured the average delay in which it took a FTD to send Confirmable packets to another FTD in a simulated smart home network. The experiments took place in a single story ADU, and the devices played the role of commercial smart plugs, such as the Eve Energy [103, 104] or Wemo [105, 106] smart plugs; as a result, the devices were placed near electrical outlets. An

[0]The official Python ASCON implementation, also known as `pyascon` [102], does *not* support variable tag length for ASCON-128. It supports variable tag length for ASCON-MAC, ASCON-PRF, and ASCON-PRFSHORT

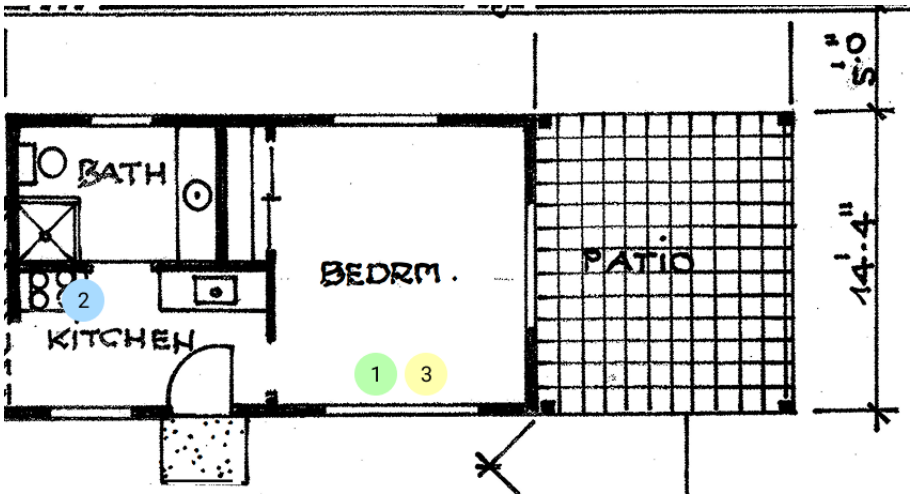


Figure 4.1: The floor plan of the single story ADU. The blue circle numbered 1 indicates the location of the delay client, and the green circle numbered 2 indicates the location of the delay server. The yellow circle with the number 3 indicates the location of the nRF52840 packet sniffer.

nRF52840 MCU was used to sniff all in-flight packets. To measure the delay, the devices were synchronized using the OpenThread network time synchronization feature [107–110]. Only FTDs were used, as it was not possible to use network time synchronization on ESP Thread Border Routers when this thesis was written [111]. The network topology was kept constant throughout all experiments.

Algorithm 1: Delay Server

```
start CoAP server;
create route to capture packets at URI delay-confirmable;

while true do
    if received request from the delay client then
        sent ← timestamp in payload of client;
        received ← current timestamp;
        delayUs ← received − sent;
        send an ACK with delayUs as payload to delay client;
    end
end
```

Algorithm 2: Delay Client

```
DelaysUs[1000] ← array to hold delays in microseconds;
socket ← socket with delay server;

i ← 0;
while i ≤ 1000 do
    sequenceNum ← i;
    networkTime ← current timestamp in microseconds;
    payload ← payload with sequenceNum and networkTime;
    send request to delay server with payload;

    if i = 0
        continue;
    elif i ≤ 1000
        wait for ACK from delay server;
        if receive ACK
            DelaysUs[i] ← delay given in payload;
            i ← i + 1;
            if i = 1000
                average ← average of all delays in DelaysUs array;
                print out average to serial monitor;
        else
            do software restart and redo current trial;
end

if first trial in experiment
    numTrials ← 0;
else
    numTrials ← value saved in Non-Volatile Storage (NVS);
numTrials ← numTrials + 1;
save value of numTrials in NVS;
if numTrials < 1000
    do software restart and start next trial;
```

4.3 Confirmable Tests

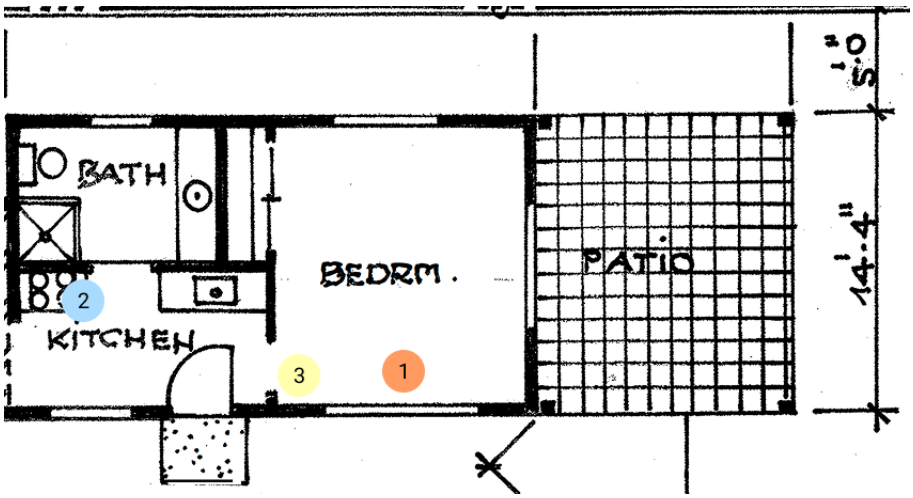


Figure 4.2: The floor plan of the single story ADU. The blue circle numbered 1 indicates the location of the FTD, and the orange circle numbered 2 indicates the location of the Border Router. The yellow circle numbered 3 indicates the location of the nRF52840 packet sniffer.

Device Number	Thread Device Type	Commercial Device
1	Border Router	Amazon Echo [54], Google Nest Hub [112]
2	FTD	Eve Energy and Wemo Smart Plug[2] [103–106]

Table 4.1: Mappings between devices used in the Throughput Confirmable experiments to the corresponding commercial devices.

The Throughput Confirmable experiments measured the average throughput for which it took a FTD to send Confirmable requests in a *tight loop* to a Border

[1]The Eve Energy [103,104] and Wemo Smart Plug [105,106] are outlet plugs that enables users to remotely toggle the electricity flow to the electronic devices connected to them, effectively powering them on or off.

Router. The network topology used is similar to the one used in the Delay experiment, but the ESP Thread Border Router is used in place of the FTD which acted as the server. The Border Router and FTD were mapped to commercial devices presently available in the market. The device number in the topology shown in Figure 4.2 maps to the row of the same number in Table 4.1. The Packet Loss Confirmable experiments measured the average packet loss when a FTD sent Confirmable requests to the Border Router in a *tight loop*.

The same topology used in the Throughput and Packet Loss Confirmable experiments, and was kept exactly the same throughout all experiments. The implementation for the Border Router is the same in both, but the URI is named `throughput-confirmable` and `packet-loss-confirmable` in the Throughput and Packet Loss Confirmable experiments, respectively. Each Confirmable experiment consisted of 1000 trials.

Algorithm 3: Border Router - Confirmable Tests

```
start CoAP server;
create route to capture packets at given URI;

while true do
    if received request from FTD then
        send an ACK to the FTD;
    end
end
```

Algorithm 4: FTD - Throughput Confirmable

```
if first trial in experiment
    numTrials ← 0;
    save value of numTrials in NVS;
socket ← socket with border router;
packetsAcked ← 0;
totalBytes ← 0;
startTime ← timestamp of current time;

while packetsAcked < 1000 do
    payload ← random sequence of 4 bytes;
    send request with payload to border router;

    if ACK has not been received
        do software restart and redo current trial;
    else
        if packetsAcked < 1000
            payload ← payload of CoAP request;
            packetsAcked ← packetsAcked + 1;
            totalBytes ← totalBytes + length of payload;
            if packetsAcked = 1000
                endTime ← timestamp of current time;
                usElapsed ← endTime − startTime;
                throughputUs ← totalBytes / usElapsed;
                Print throughput to serial monitor;
                break;
end

numTrials ← numTrials + 1;
save value of numTrials in NVS;
if numTrials < 1000
    do software restart and start next trial;
```

Algorithm 5: FTD - Packet Loss Confirmable

```
if first trial in experiment
    numTrials ← 0;
    save value of numTrials in NVS;
socket ← socket with border router;
numAcked ← 0;

while true do
    payload ← random sequence of 4 bytes;
    send request with payload to border router;

    if ACK has been received
        numAcked ← numAcked + 1;
        if numAcked = 1000
            packetLoss ← (1000 − numAcked) / 1000;
            print packetLoss ratio to serial monitor (which will be 0);
            break;
    else
        packetLoss ← (1000 − numAcked) / 1000;
        print packetLoss ratio to serial monitor;
        break;
end

numTrials ← numTrials + 1;
save value of numTrials in NVS;
if numTrials < 1000
    do software restart and start next trial;
```

4.4 Observe Tests

The Throughput Observe experiments measured the average throughput for which it took a FTD to send Non-Confirmable packets to a Border Router. The scenario used in these experiments was based on an example in RFC 7641 [41] which describe a smart thermometer hosting a RESTful endpoint that enables subscribers to receive Observe notifications of the temperature every few seconds. The simulated temperature sent by the FTD in the experiments was a random room temperature value, between 68-74 °F (inclusive) [113]. The Packet Loss Observe experiments measured the extent to which *packet loss* occurs among the Non-Confirmable Observe notifications sent by the FTD. The Throughput and Packet Loss Observe experiments utilized the same topology as the Confirmable experiments shown in Figure 4.2. Unlike the Confirmable experiments, each Observe experiment consisted of 100 trials.

Algorithm 6: Border Router - Throughput Observe

```
if first trial in experiment
    numTrials ← 0;
    save value of numTrials in NVS;
socket ← socket with FTD;
numReceived ← 0;
totalBytes ← 0;
startTime ← timestamp of current time;

send observe request to FTD to subscribe to temperature endpoint;
if failed to subscribe to temperature endpoint
    do software restart and redo current trial;

expectedTotalBytes ← 1000 requests · 8 bytes of payload;
while true do
    if receive CoAP observe notification
        numReceived ← numReceived + 1;
        totalBytes ← totalBytes + length of payload;

        if totalBytes = expectedTotalBytes
            endTime ← timestamp of current time;
            throughputUs ← totalBytes / (endTime − startTime);
            Print the throughput to the serial monitor in us and ms;
            break;
end

send cancellation request to FTD;
numTrials ← numTrials + 1;
save value of numTrials in NVS;
if numTrials < 1000
    do software restart and start next trial;
```

Algorithm 7: FTD - Throughput Observe

```
start CoAP server;
create route to capture packets at URI temperature;

subscribed ← false;
token ← 0;
sequenceNum ← 0;

while true do
    if received subscription request from border router
        subscribed ← true;

        token ← generate random token to represent observe subscription;
        temperature ← random temperature in between 68-74°F;
        send ACK with temperature and sequenceNum as payload;

        wait for 1 second;
    elif subscribed = true
        if received cancellation request from border router
            Cancel subscription with token value token;
            subscribed ← false;
        else
            temperature ← random temperature in between 68-74°F;
            sequenceNum ← sequenceNum + 1;
            payload ← temperature and sequenceNum;
            send notification with payload to border router;

            wait for 1 second;
end
```

Algorithm 8: Border Router - Packet Loss Observe

if first trial in experiment
    numTrials $\leftarrow 0$;
    save value of `numTrials` in NVS;

socket $\leftarrow$ socket with FTD;
numReceived $\leftarrow 0$;

send observe request to FTD to subscribe to `temperature` endpoint;
if failed to subscribe to `temperature` endpoint
    do software restart and redo current trial;

while true **do**
    if receive Non-Confirmable notification
        numReceived $\leftarrow$ numReceived $+ 1$;
    elif receive Confirmable packet
        packetLoss $\leftarrow \frac{1000 - \text{numReceived}}{1000}$;
        Print `packetLoss` to serial monitor;
        break;
end

send cancellation request to FTD;
numTrials $\leftarrow$ numTrials $+ 1$;
save value of `numTrials` in NVS;
if numTrials < 1000
    do software restart and start next trial;

Algorithm 9: FTD - Packet Loss Observe

```
start CoAP server;
create route to capture packets at URI temperature;

subscribed ← false;
token ← 0;
sequenceNum ← 0;

while true do
    if received subscription request from border router
        subscribed ← true;
        token ← generate random token to represent observe subscription;
        temperature ← random temperature in between 68-74°F;
        send ACK with temperature and sequenceNum as payload;
        wait for 1 second;

        for sequenceNum = 1 to 1000 do
            temperature ← random temperature in between 68-74°F;
            payload ← temperature and sequenceNum;
            send notification with payload to border router;
            wait for 1 second;
        end

        payload ← temperature and sequenceNum;
        send Confirmable notification with payload to border router;
    elif subscribed = true
        if received cancellation request from border router
            Cancel subscription with token value token;
            subscribed ← false;
end
```

4.5 Energy Consumption Tests

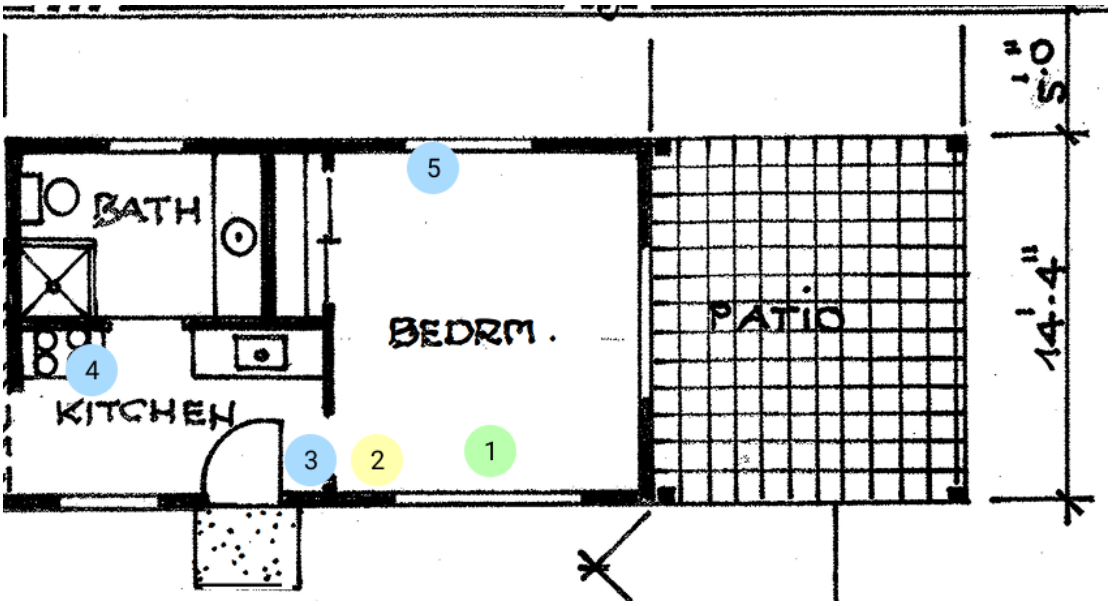


Figure 4.3: The floor plan of the ADU. The green circle numbered one corresponds to the location of the Border Router. The yellow circle numbered two corresponds to the location of the nRF52840 802.15.4 sniffer. The blue circles labelled 3, 4, and 5, corresponds to the locations of the sensor for the front door, the air quality monitor, and the window sensor, respectively.

The Energy Consumption experiments measured the average energy usage of SEDs in a simulated smart home network. The SEDs acted as battery powered devices that were in deep sleep, only waking up to send data to the Border Router. The experiments took place in the same single story ADU as the network performance experiments, and consisted of three ESP32-H2s functioning as SEDs, and an ESP Thread Border Router. The devices were mapped to commercial Thread devices, as shown in Table 4.2. The devices will be referred to by the specific

numbers they are identified as in the table, and placed in the ADU with respect to their roles.

Device Number	Thread Device Type	Commercial Device
1	Border Router	Amazon Echo[3] [54], Google Nest Hub[3] [112]
3	SED	Eve Motion[4] [104, 114]
4	SED	Eve Room[5] [104, 115]
5	SED	Door & Window Sensors[6] [104, 116, 117]

Table 4.2: Mappings between the devices used in the experiments to the corresponding commercial devices.

The workload of the SEDs simulated the network traffic of a real-life smart home in accordance to the role they were assigned in Table 4.2. SEDs (3) to (5) functioned as battery powered sensors and only sent packets in the following two scenarios:

- **Scenario 1:** The SEDs sent packets to border router (1) to report the status of their battery life exactly *once* per day.
- **Scenario 2:** The SEDs sent packets informing border router (1) whenever an event has occurred.

[3]The 4th Generation Amazon Echo [54] and Google Nest Hub Max and 2nd Gen [112] support Thread border router functionality.

[4]The Eve Motion sensor [114] is a battery powered indoor and outdoor motion sensor.

[5]The specific corresponding products are the Aqara Door and Window Sensor P2 [116] the Eve Door and Window sensor [104, 117].

[6]The Eve Room is a sensor used to report the temperature and air quality inside a home [104, 115].

These were based on Betzler et al. [118], which evaluated the performance of Zigbee networks: the researchers state that every smart home network contains two types of in-flight packets: packets which are sent in either a *periodic* or *aperiodic* manner. Scenarios 1 and 2 corresponds to packets that were sent periodically and aperiodically, respectively.

Each experiment simulated 365 days, or 1 year's, worth of smart home network activity in a short period of time. The timescale in the experiments were compressed by equating 30 seconds to represent 1 day's worth of smart home network activity. To simulate 1 year's worth of smart home network activity when a single day is compressed to be 30 seconds long:

$$30 \text{ second } \cdot \ 365 \text{ days} = 10950 \text{ seconds} \tag{4.1}$$

$$10950 \text{ seconds } \div \ 60 \text{ minutes} = 182.5 \text{ minutes} \tag{4.2}$$

Each experiment was set to have a duration of exactly $\lceil 182.5 \rceil = 183$ minutes.

Since 1 day's worth of smart home network activity was simulated in 30 seconds, and Scenario 1 packets were sent once per day, each SED sent a Scenario 1 packet once every 30 seconds. With respect to Scenario 2, the SEDs sent such packets at different moments with respect to the roles given in Table 4.2. More specifically, each SED sent Scenario 2 packets under the following conditions:

- Suppose SED (3) was set up such that its motion sensor was enabled only when I was asleep at night, or when I am outside my home (i.e. the ADU).

The events which will cause SED (3) to activate is when there is an animal (e.g. a raccoon) which crosses the ADU in the middle of the night, or when someone (e.g. a neighbor or delivery driver) arrives at the front door when I am not home. We will assume that such occurrences happen, on average, 3 times per month. Thus, these occurrences will happen, on average:

$$12 \text{ months} \cdot 3 \text{ occurrences} = 36 \text{ times in one year} \tag{4.3}$$

In each experiment, there will be a total of 36 times in one year in which SED (3) will send a Scenario 2 packet.

- When I am at home, I always open my windows to let fresh air in. When SED (4), the air quality monitor, notifies me that the outside air is of poor air quality, I will close my windows. Suppose that such notifications sent by SED (4) happen, on average, 10 times a year. Thus, there will be a total of 10 times when SED (4) will send a Scenario 2 packet in each experiment.

- Suppose SED (5) has been set up next to the window in the bedroom of the ADU, in order to detect when the window was open, but should be closed. For example, if the sun has already set, SED (5) will send a notification to remind me to close the window. There may be moments in which I am not home and would like to use my smartphone to double-check whether the window is open or closed. We will suppose that the window sensor will

inform me that the window is open, or send me the status of whether the window is open (when I query for it), approximately 12 times a year. As a result, SED (5) will send Scenario 2 packets exactly 12 times a year in each experiment.

The pseudocode for Border Router (1) is shown in Algorithm 10. The Border Router hosted two CoAP endpoints: `battery` and `event`, to handle packets sent by the SEDs under Scenarios (1) and (2), respectively. The pseudocode of the application layer program running in SEDs (3) to (5) is shown in Algorithm 11. All SEDs ran the same program, but the variable `NUM_EVENTS` was set to 36, 10, and 12 for SEDs (3), (4), and (5) respectively, following the scenarios described in the bullet points.

Algorithm 10: Border Router Implementation

```
start CoAP server;
create route to capture Scenario (1) packets at URI battery;
create route to capture Scenario (2) packets at URI event;

while true do
    if received packet at URI battery or event then
        send a CoAP ACK back to the sender;
    end
end
```

Algorithm 11: Workload of the SEDs

```
NumEvents ← number of Scenario (2) packets to send;
wakeup ← current timestamp;
if The device just woke up from deep sleep then
    deviceId ←
     get Universally Unique Identifier (UUID) of device from NVS;
    wakeupNum ← get number of wakeups done from NVS;
end
else
    deviceId ← generate UUID for device;
    // Device powering on counts as first "wakeup".
    wakeupNum ← 1;
end
socket ← socket with border router;

interval ← ⌊365 ÷ NumEvents⌋;
if wakeupNum (mod NumEvents) ≡ 0 then
    eventOccurred ← true;
    payload ← eventOccurred and deviceId;
    send POST request with payload to border router at URI event;
end
batteryLife ← 100;
payload ← batteryLife and deviceId;
send POST request with payload to border router at URI battery;

wait to receive ACK for battery packet;
if sent an event packet then
    wait to receive ACK for event packet;
end
current ← current timestamp;
deepSleepDuration ← 30 seconds − (current − wakeup);
Go to deep sleep for "deepSleepDuration" seconds;
```

4.6 Independent and Dependent Variables

The independent variables for the Delay, Throughput Confirmable, Throughput Observe, Packet Loss Confirmable, Packet Loss Observe, and Energy Consumption experiments were:

- The AEAD algorithm used in OpenThread: AES-CCM, ASCON-128a, ASCON-128, or no encryption algorithm.
- The Transmit (TX) power used by the devices: 20 dBm, 9 dBm, or 0 dBm. These three TX powers were chosen, as these TX power values are used in the ESP32-H2 datasheet[7] [65] when discussing the energy consumption of the device. In all experiments, every device was set to use the same TX power.

The dependent variables in each of the experiments were as follows:

- The dependent variable for the Delay experiments was the average delay, calculated in milliseconds.
- The dependent variable for the Throughput Confirmable and Observe experiments was the average throughput measured in bytes per second.

[7]The TX power value of -24 dBm is used in the ESP32-H2 datasheet when discussing the energy consumption of the device. However, the experiments were not able to run at -24 dBm, as the TX power was too small: the ESP32 Thread devices were unable to communicate with each other at such a low TX power value.

- The dependent variable for the Packet Loss Confirmable experiments was the average packet loss percentage calculated in the 1000 trials that were run in each experiment. The packet loss percentage of each trial was calculated from the ratio between the number of ACKs sent by the Border Router and the total number of requests sent by the FTD.

- The dependent variable for the Packet Loss Observe experiments was the average packet loss percentage calculated in the 100 trials that were run in each experiment. The packet loss percentage of an individual trial was calculated from ratio between the number of notifications received by the Border Router, over the number of notifications sent by the FTD (i.e. 1000 notifications) in the trial.

4.7 Test Environment

4.7.1 Network Performance Experiments

The testbed for the Delay, Throughput Confirmable, Throughput Observe, Packet Loss Confirmable, and Packet Loss Observe experiments consisted of a Dell Optiplex 9020 [121, 122] desktop running Ubuntu that controlled, monitored,

[8]The picture for the nRF52840 development kit came from the Nordic Semiconductors nRF52840 Product Brief [88]. The pictures of the ESP32-H2 and ESP Thread Border Router came from the respective Amazon webpages for these two devices [119, 120].

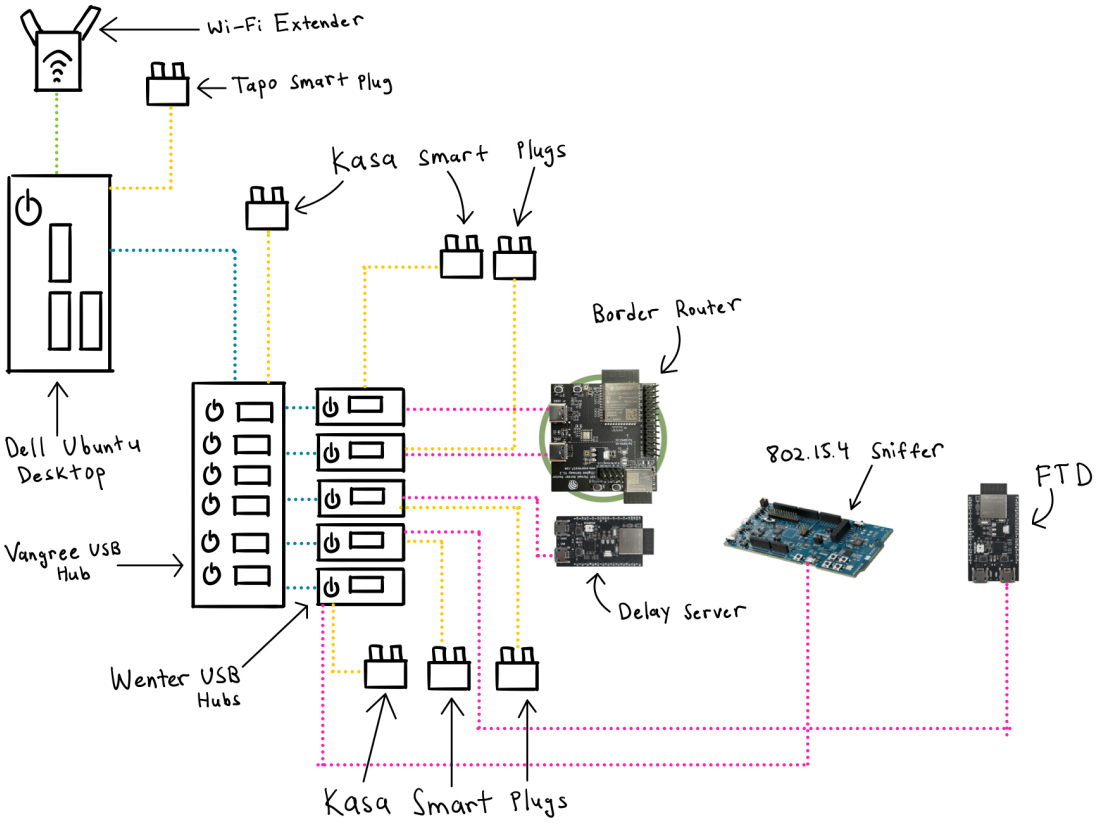


Figure 4.4: The experimental testbed[8] used to run the Delay, Throughput, and Packet Loss experiments. The blue dotted lines indicate a USB connection which enables only data transfer, the yellow dotted lines indicate a wired connection supplying AC power, the pink dotted lines indicate a USB connection which supports *both* data and power transfer, and the green dotted lines indicate an Ethernet connection.

and flash programs[9] to the devices used in the experiments. A daisy-chained circuit of USB hubs and cables connected the desktop computer to each device.

[9]This setup could theoretically allow the Ubuntu desktop to flash programs to the nRF52840 microcontroller. Throughout all experiments, the Ubuntu desktop only flashed to the ESP32 devices, and *not* to the nRF52840 sniffer. The nRF52840 sniffer was always running the nRF 802.15.4 sniffer [89] program, which had already been flashed to the device before running any experiments.

The central hub of the daisy-chained circuit was an externally powered Vangree 17-port USB 3.0 hub [123].

Only the devices required by the experiment were powered on and placed throughout the ADU in accordance to Figures 4.1, 4.2, and 4.3.. The nRF52840 packet sniffer was used in all experiments to capture all packets. Each device was connected to a Wenter 5-port USB hub [124], which was connected to the Vangree USB hub to enable data transfer between the Ubuntu desktop and the device. To control whether a device was powered on or off, the external power supply of each Wenter USB hub was connected to a TP-Link Kasa EP10 smart plug [125]. Long extended USB cables were used to connect the device to their respective USB hub.

Every experiment was run in an automated manner. The desktop Python and Bash automation scripts that started and stopped an individual Throughput, Packet Loss, or Delay experiment. At the end of each experiment, the Python scripts uploaded all logs of serial monitor outputs to a Synology NAS. GitHub Action workflows used to remotely instruct the Ubuntu desktop to start an experiment, along with printing out the status of a currently running experiment.

4.7.2 Energy Consumption Experiments

4.7.2.1 Power Profiler Kit II (PPK2)

The Nordic Semiconductors Power Profiler Kit II (PPK2) was used to measure the current of SED (3) in the Energy Consumption experiments. The PPK2 measures current consumption in milliampere (mA) for a set period of time (or even indefinitely), and is endorsed by Espressif for measuring the deep sleep current of ESP32 microcontrollers [126]. SED (3) was connected to the PPK2 by using the red, black, and brown jumper wires shown in Figure 4.5, and a micro USB cable connected the device the Ubuntu Desktop, which ran the nRF Connect Power Profiler application. The Power Profiler application displayed a live view of the current being measured in real time. SED (3) was powered by the power supply provided by the PPK2 at 3.3V. At the end of each experiment, the data recorded by the PPK2 was exported as a `.ppk2` file. During post-processing, each PPK2 file was opened in the PPK2 software and exported as a `.csv` file, which is then used to calculate average energy consumption in milliampere-hours (mAh).

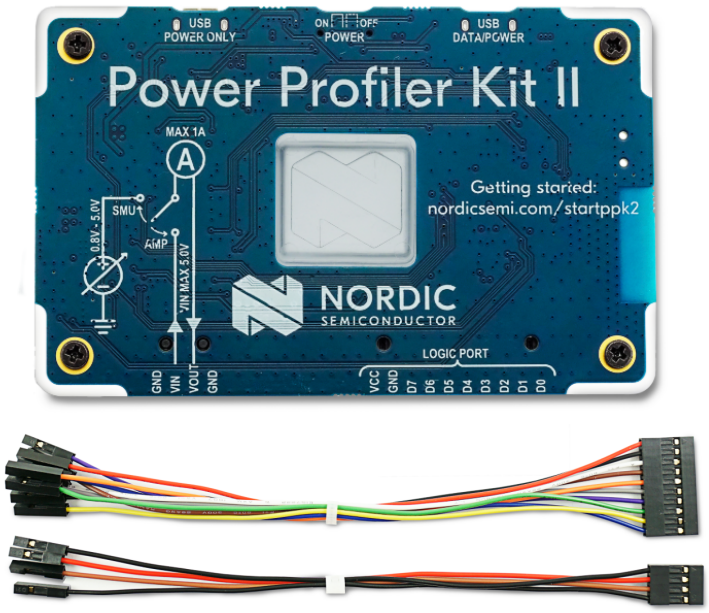


Figure 4.5: A picture of the PPK2 (top) and jumper cables (bottom) [127].

4.7.2.2 ESP32-H2 J5 Header

An ESP32-H2 contains a **J5 header** jumper that must be removed to enable a measurement device to accurately record its energy consumption; if not removed, a measurement device will not be able to accurately measure the current for the device. When the J5 jumper is *closed*, the PPK2 measures the deep sleep as consuming an average of ~ 1.21 mA. When the J5 jumper is *removed*, the PPK2 will record the ESP32-H2 average deep sleep current as $\sim 7.5 - 8$ microampere (uA), closer to the expected deep sleep current of ~ 7 uA of energy on deep sleep [65, 128].Tthe ESP32-H2 *must not* be supplied with 5V of power when the J5 jumper is *removed*, as doing so could risk damaging the device [129]; it must

instead be supplied with 3.3V of power. Thus, PPK2 was set to power SED (3) with 3.3V in the every experiment.

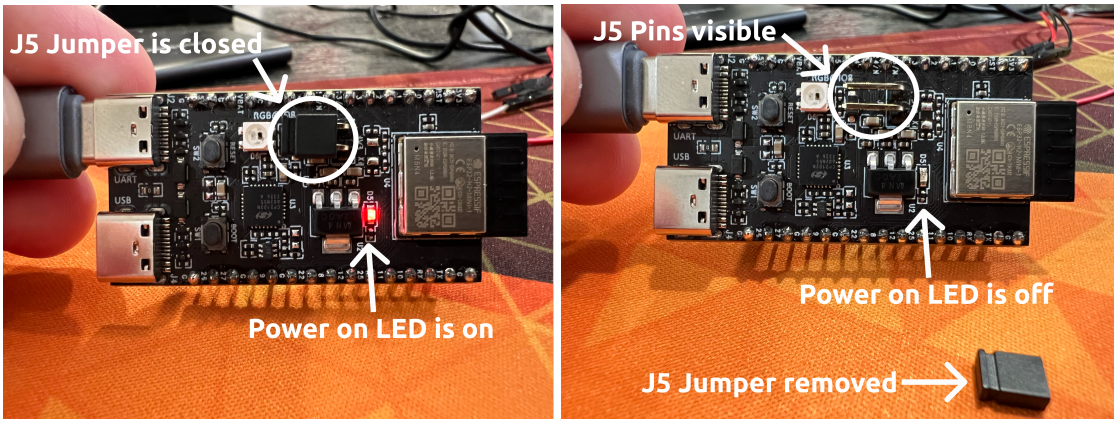


Figure 4.6: The ESP32-H2 when the J5 jumper it was removed, and when covering the J5 header pins. The power-on LED is off when the J5 jumper is not plugged in, even though the device itself was still on (powered by USB).

4.7.2.3 Limitations

The Energy Consumption experiments were run in a *manual* fashion, the process of opening and closing J5 jumper from SED (3) could not be automated. Instead, each experiment was manually started and stopped, along with monitoring and flashing of the SEDs and Border Router with the driver programs [97, 98] set the independent variables needed for current experiment. Since the experiments were conducted manually, it made it difficult to run multiple trials. Instead, only one trial was run for each experiment.

In each experiment, the PPK2 measured the energy consumption of SED (3)

for 190 minutes, instead of 183 minutes. SED (3) was always powered off at the start of an each experiment, and the Power Profiler user interface is used to manually instruct the PPK2 to start recording the current, and power on the SED, in this exact order. If the PPK2 was set to measure the current for exactly 183 minutes, then it would only measure the device when powered on for slightly *less than* 183 minutes. To guarantee that the PPK2 measures the power consumption of the SED after it has been powered on for *at least* 183 minutes, the PPK2 was set to measure the energy consumption for 190 minutes. In post-processing, the results were gathered based on the energy consumption samples starting when SED (2) was powered on, up until 183 minutes after the moment when the SED first powered on.

4.8 Experimental Setup

4.8.1 Network Performance Tests

Section 4.7 describes the most developed version of the automation testbed that created for the Network Performance experiments. However, the testbed was iteratively improved between experiments. As a result, the experiments were run on different testbed configurations. The configurations for each experiment were:

- **Configuration 1:** A Lenovo Yoga 730-15IWL [130] laptop running Ubuntu was used, rather than the Dell Optiplex 9020 desktop computer. GitHub Action workflows were not yet set up. Bash scripts were run start the experiments, and the experiment data were manually uploaded to the Synology NAS after each experiment. The nRF52840 802.15.4 sniffer was placed near the Border Router. **All Delay experiments were run under Configuration (1)**.

- **Configuration 2:** The Dell Ubuntu desktop was used and ran a set of Python scripts were created to run alongside the Bash scripts in automating the experiments. Only a single command was needed to run the experiments in a completely automated manner, but GitHub Action workflows were not yet used. The nRF52840 802.15.4 packet sniffer was moved from the table containing the Ubuntu laptop/desktop, to the middle of the room, near

the door. **All Throughput Confirmable experiments where the TX power was** 20 **dBm experiments ran under Configuration (2)**.

- **Configuration 3:** The automation testbed shown in Figure 4.8. **All Throughput and Packet Loss Observe experiments ran under Configuration (3)**.

The location of the device operating client in the Delay experiments, and the FTD in the Confirmable and Observe experiment, remained the same in all configurations[10].

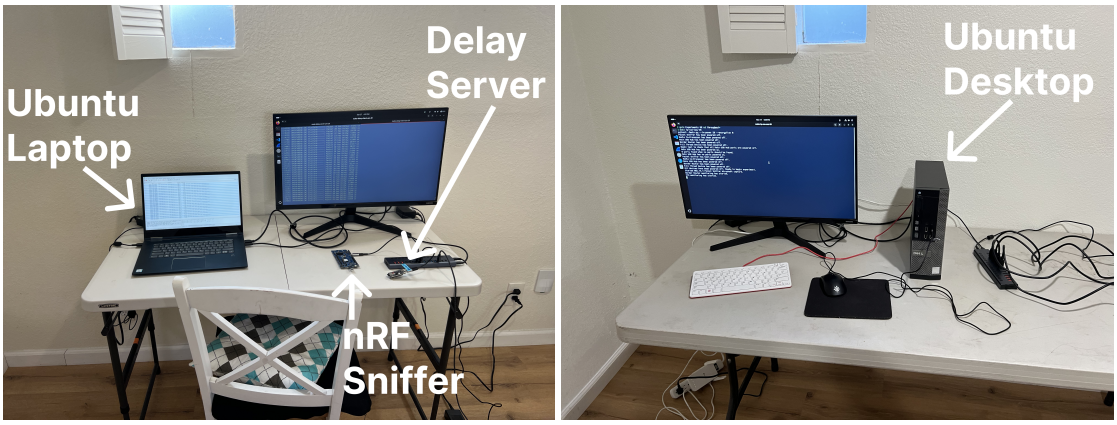


Figure 4.7: Shown left and right are the setup of Configuration (1) and (2), respectively.

[10]The delay server was present in the most developed version of the testbed shown in Figure 4.8, as the Delay experiments were rerun with this configuration. However, before doing such reruns, I had made modifications to the source code [100], which introduced new bugs and made the results of the reruns invalid. At the time, the Observe experiments were not yet run, so I chose to discard these results and used the initial, valid results I had for the Delay experiments.

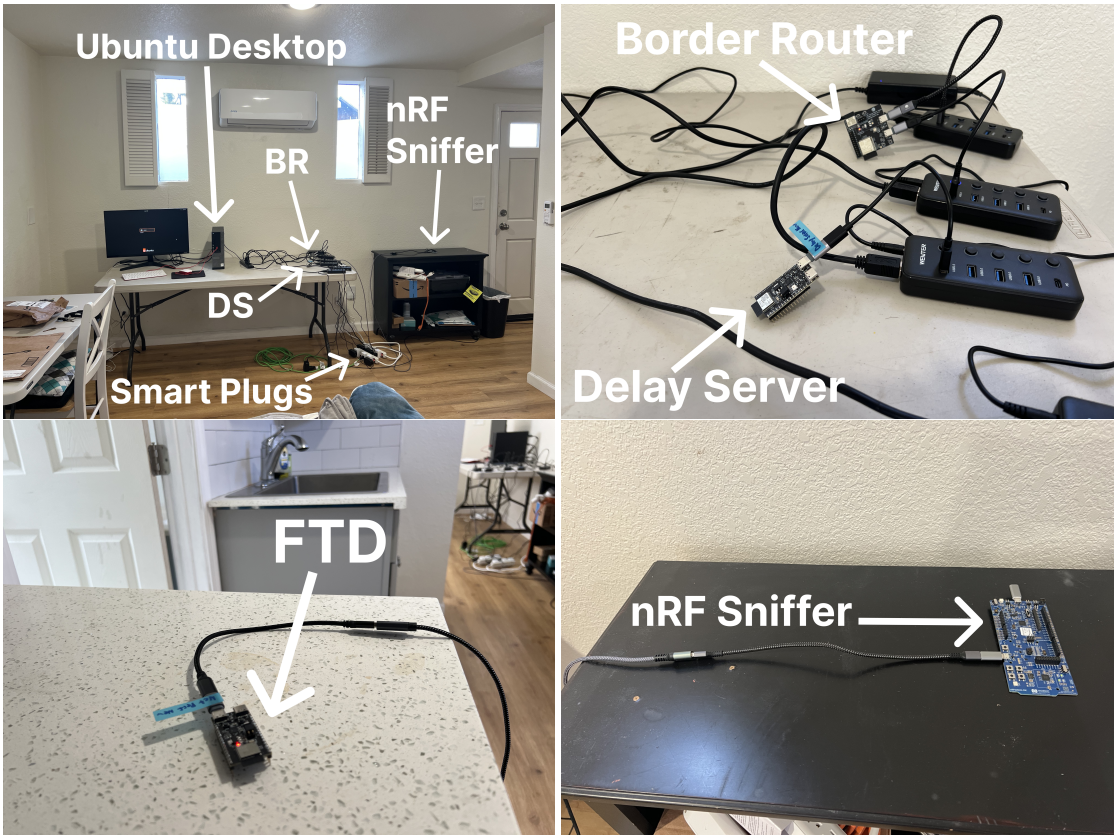


Figure 4.8: The locations of the FTD, nRF52840 802.15.4 packet sniffer, Border Router, and Delay Server in Configuration (3). The abbreviations "BR" and "DS" are short for "Border Router" and "Delay Server", respectively. The FTD remained in the same location throughout the evolution of the testbed, and operated as the delay client in the Delay experiments.

4.8.2 Energy Consumption Tests



Figure 4.9: The experimental setup for the Energy Consumption experiments. SED (4) is not show in this picture, as it is hidden behind the wall past the door.

The Border Router, the SEDs, the nRF52840 802.15.4 sniffer, and PPK2 were all connected to the Ubuntu desktop using long USB cables. The Ubuntu desktop was running the Power Profiler software and Wireshark, along with monitoring the serial monitor output of the Border Router, and the UART[11] outputs of SEDs (4) and (5). To run the experiments manually, I developed and followed a series to steps which consisted of powering on the devices, setting up the Ubuntu desktop to record the serial monitor and UART outputs of the devices, setting up the PPK2 to record the current of SED (3), and uploading all logs and data recorded by the PPK2 to my Synology NAS. The Kasa smart plugs and the Wenter USB hubs were not used in these experiments. The Vangree USB hub was still used to connect the SEDs, Border Router, nRF sniffer, and PPK2 to the Dell Ubuntu desktop, as the desktop itself had a limited number of USB ports.

[11]The SEDs were connected to the USB cables via UART [131], as the SEDs kept crashing

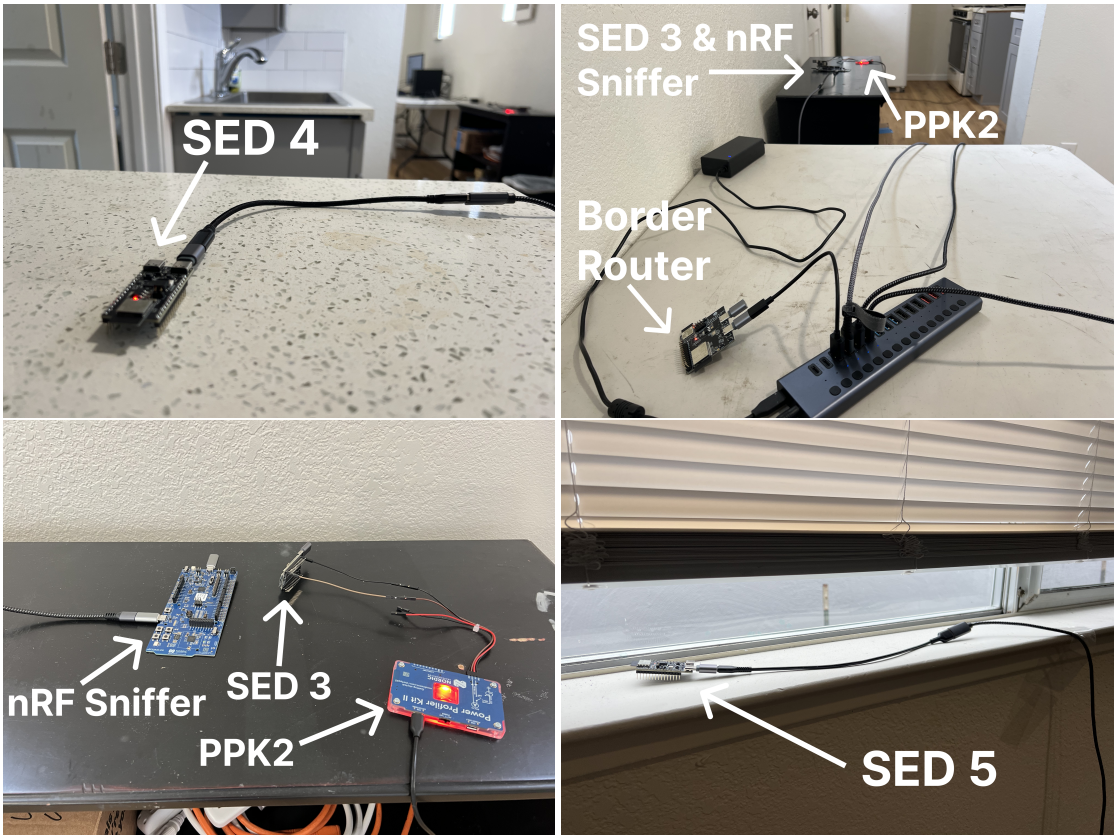


Figure 4.10: The Border Router, the SEDs, the nRF52840 802.15.4 sniffer, and the PPK2.

when they were flash and monitored through USB serial.

Chapter 5

Results

This Chapter presents the results gathered from the experiments. The results of the Correctness Tests showed that the modified implementation of OpenThread is properly secured by ASCON AEAD. The results of the Network Performance and Energy Consumption experiments respectively displayed no significant difference in network performance and energy usage of ESP Thread devices when LibAscon AEAD is used instead of AES-CCM.

5.1 Correctness Tests

Following the instructions described in Section 4.1, I was able to successfully decrypt all sample ASCON-128a and ASCON-128 ciphertext payloads that were sent and received between the ESP32 Thread devices. Thus, the Correctness Tests

have proven than my modified implementation of OpenThread is properly secured by the LibAscon implementations of ASCON-128a and ASCON-128. As stated in Section 4.1, the `C` programs that were run, along with the sample ciphertexts that were used, are all publicly available on GitHub [101].

5.2 Network Performance Tests

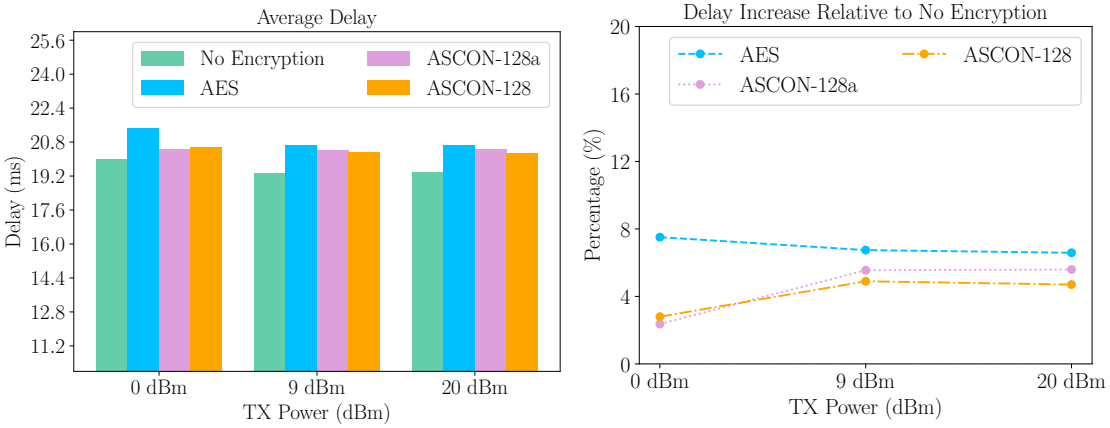


Figure 5.1: The results of the Delay experiments.

The results of the Delay and Throughput experiments are shown in Figures 5.1 and 5.2, respectively. The most salient differences in network performance are exhibited in the Delay experiments, where the highest increase in average delay was a slightly less than 8% increase when AES-CCM was used at a TX power of 0 dBm, compared to when plaintext communications was used instead. The average throughput observed between the different encryption algorithms did not decrease by more than 5% relative to plaintext communications. The differences

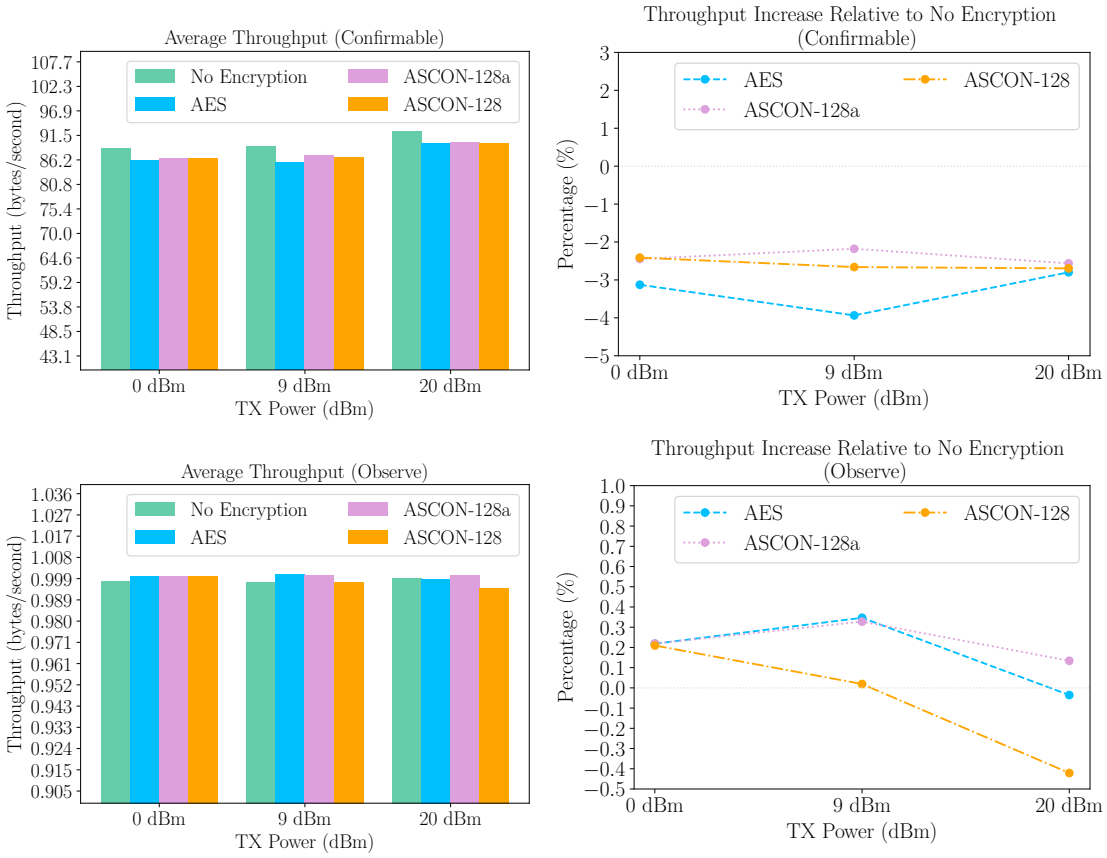


Figure 5.2: The results of the Throughput experiments.

are less noticeable in the Throughput Observe experiments, where the difference never exceeded 1%. As shown in Figure 5.3, there was no significant difference in packet loss when the encryption algorithm was varied; such differences never exceed 1%, let alone 0.001000%. No packet loss was found in any Packet Loss Confirmable experiment. This is to be expected, as the use of Confirmable CoAP requests requires every request sent by the FTD to be acknowledged by the Border Router.

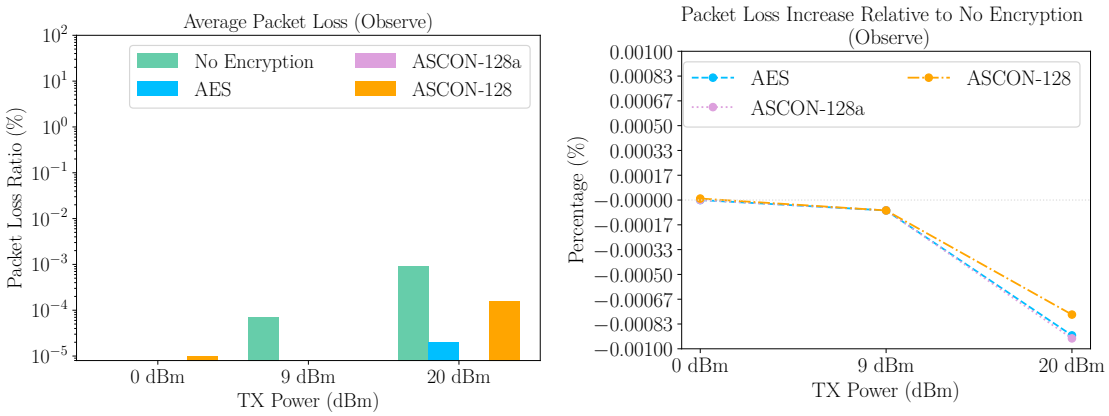


Figure 5.3: The results of the Packet Loss Observe experiments.

No major difference was observed in delay, throughput, and packet loss, respectively even when the encryption was changed or plaintext communications were used. Therefore, the use of LibAscon does not significantly impact network performance. Such results indicate the feasibility for commercial ESP32 Thread devices to use a version of OpenThread secured under ASCON without having their functionality adversely affected.

5.3 Energy Consumption Tests

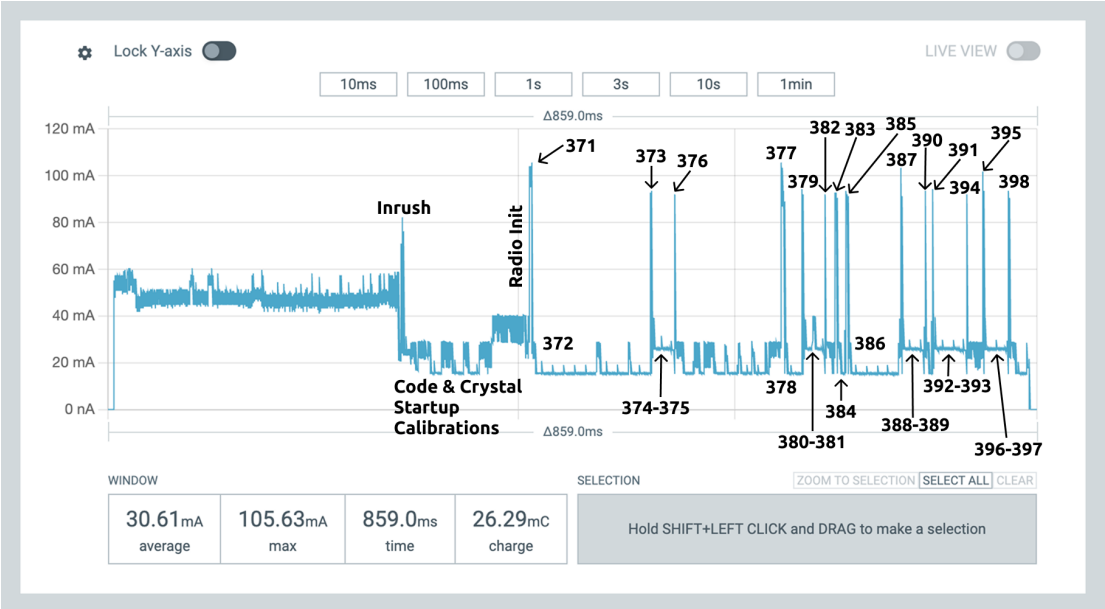
This section begins with an analysis of the energy consumption waveforms of a SED over a single wakeup, followed by a discussion of the predicted differences in battery life when ASCON AEAD is used, compared to AES-CCM. Such analysis is used to form a *hypothesis* for the Energy Consumption tests, which is then examined against the results of the experiments.

5.3.1 Waveform Analysis

Figure 5.4 displays the energy consumption waveform diagram for a SED that sent one Scenario (1) and Scenario (2) packet each, operating at 20 dBm. The waveform diagram was recorded on the PPK2 at a sampling rate of 100k samples per second. Using the waveform diagram given in the Thread Battery Operated Devices whitepaper [45] as a template, the peaks and valleys in Figure 5.4 were labelled with the corresponding activity of the SED that generated the given spike in energy consumption. The Wireshark packet capture for all packets that exchanged between the SED and Border Router is shown in Figure 5.4. The peaks and valleys in Figure 5.4 are labelled with the number of packet, or group of packets, numbered by Wireshark.

When the encryption algorithm was varied, the only difference that can occur is duration for encrypting a packet before transmission, and decrypting a packet after reception. Thus, when changing in the encryption algorithm, the differences in energy consumption can only occur when the SED is *on wakeup* and not on deep sleep. The waveform diagrams of the SED using ASCON-128a, ASCON-128, and plaintext communications are not shown[1], as they look similar to the waveform diagram in Figure 5.4.

[1]The waveform diagrams and corresponding packet captures when the SED is using ASCON-128a, ASCON-128, and plaintext communications are shown in Appendix Chapter C.



371	246.491384	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	98	Child Update Request
372	246.494078			IEEE 802.15.4	31	Ack
373	246.604168	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
374	246.605263			IEEE 802.15.4	31	Ack
375	246.622999	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	118	Child Update Response
376	246.626332			IEEE 802.15.4	31	Ack
377	246.724873	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	CoAP	102	CON, MID:42724, POST, TKN:87 e6, /event
378	246.727695			IEEE 802.15.4	31	Ack
379	246.744234	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
380	246.745328			IEEE 802.15.4	31	Ack
381	246.763054	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	CoAP	78	ACK, MID:42724, 2.03 Valid, TKN:87 e6, /event
382	246.765108			IEEE 802.15.4	31	Ack
383	246.774220	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	87	Data Request
384	246.776563			IEEE 802.15.4	31	Ack
385	246.784178	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	CoAP	104	CON, MID:42725, POST, TKN:cc 81, /battery
386	246.787065			IEEE 802.15.4	31	Ack
387	246.835105	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
388	246.836199			IEEE 802.15.4	31	Ack
389	246.853046	da:d7:7c:db:07:d2:86:b9	86:33:20:29:c2:19:cb:d0	6LoWPAN	152	Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29
390	246.857469			IEEE 802.15.4	31	Ack
391	246.864337	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
392	246.865431			IEEE 802.15.4	31	Ack
393	246.893069	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	122	Data Response[Malformed Packet]
394	246.896531			IEEE 802.15.4	31	Ack
395	246.910917	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
396	246.912011			IEEE 802.15.4	31	Ack
397	246.933054	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	CoAP	78	ACK, MID:42725, 2.03 Valid, TKN:cc 81, /battery
398	246.935109			IEEE 802.15.4	31	Ack

Figure 5.4: The energy consumption waveform[2] for a SED during a single wakeup, along with the corresponding Wireshark packet capture.

5.3.2 Differences in Battery Lifetime

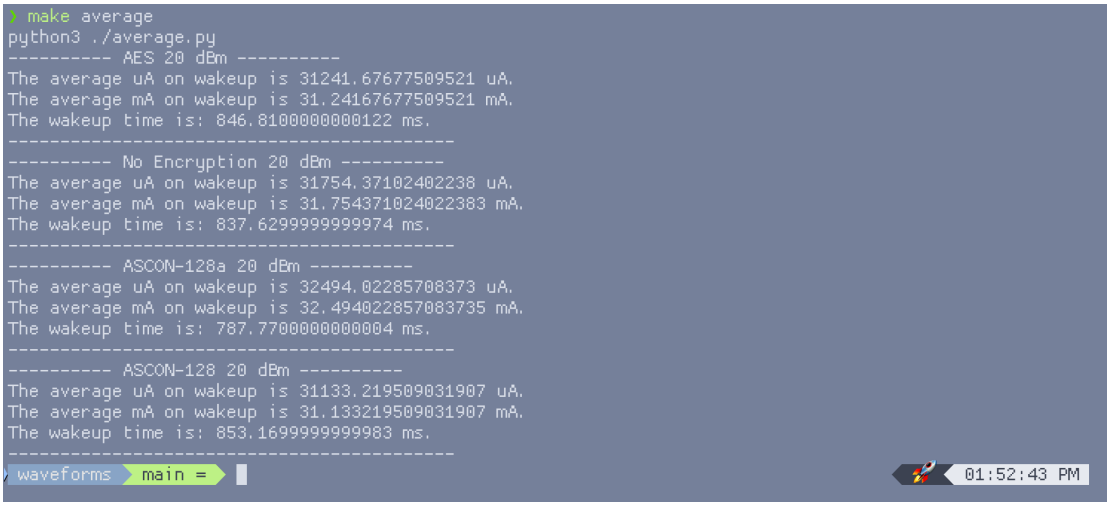


Figure 5.5: The average energy consumption of the SED in milliampere (mA), under the different encryption algorithms (along with when none is used).

The energy consumption waveforms of a SED which sent one Scenario (1) and (2) packet each on a single wakeup was recorded[3] when using AES-CCM, ASCON-128a, ASCON-128, and plaintext communications. A Python script was used to obtain the average energy consumption of the SED *during wakeup*[4] from the waveform diagrams, and its output is displayed in Figure 5.5. The duration that the SED was on wakeup in each of the four diagrams is also shown. In Figure 5.5, the highest average energy consumption on wakeup was when the SED was secured under ASCON-128a. To obtain the worst case estimate on the difference in battery life when the encryption algorithm was changed, the battery life of the

[3]The waveforms used for the calculations in this section are the ones shown in 5.4 and in Appendix C.

SED when secured by ASCON-128a will be compared to AES-CCM.

5.3.2.1 Assumptions

Suppose that the SED is a battery powered smart home device that wakes up once per day sends to Scenario 1 packet on wakeup to report of its battery life – and an event always occurs when the SED wakes up, a Scenario 2 (event) packet being sent in the same wakeup[5]. Furthermore, suppose that the duration of the wakeup of the SED is exactly 1 second long, regardless if it was using plaintext communications or an encryption algorithm[6].

5.3.2.2 Yearly Energy Usage Under AES-CCM

Since the SED wakes up once per day, where wakeup duration is 1 second long, the SED will be on wakeup for a total of 365 seconds per year. In terms of hours, the SED is on wakeup for:

$$(365 \text{ seconds} \div 60 \text{ seconds}) \div 60 \text{ minutes} = 0.1013888889 \text{ hours} \tag{5.1}$$

[4]The `ppk2` files containing the data for each of the waveforms were converted to `csv` files. The Python script was run on the `csv` files to calculate the average energy consumption on wakeup by taking the average of samples with a current greater than or equal to 9 uA. Even though the deep sleep current of an ESP32-H2 had its energy consumption at 7 uA [65], in practice, the observed deep sleep current can be as high as 8.84 uA. A slightly higher threshold of 9 uA ensured that currents samples not generated from deep sleep were calculated.

[5]This simplifying assumption is made, as Figure 5.5 shows that the energy consumption of a SED when Scenario (1) and (2) packets are sent within a single wakeup.

[6]The wakeup durations shown in Figure 5.5 displays that the wakeup time of a SED is always less than 1 second when the device is sending both a Scenario 1 and 2 packet, regardless of the encryption algorithm being used, or none is used at all. A 1 second wakeup time gives an upper bound on the wakeup duration of a SED.

The average energy consumption of the SED under ASCON-128a shown in Figure 5.5 was 31.24167677509521 mA. The energy consumption in milliampere-hours (mAh)[7], when the SED is on wakeup during the course of one year, is:

$$31.24167677509521 \text{ mA} \cdot 0.1013888889 \text{ hours} = 3.167558896 \text{ mAh} \tag{5.2}$$

There is approximately:

$$((365 \text{ days} \cdot 24 \text{ hours}) \cdot 60 \text{ minutes}) \cdot 60 \text{ seconds} = 31536000 \text{ seconds} \tag{5.3}$$

in one year. The SED is one wakeup for 365 seconds in one year. Thus, the SED will be in deep sleep for:

$$31536000 - 365 = 31535635 \text{ seconds} \tag{5.4}$$

in the course of one year. A time period of 31535635 seconds is equivalent to:

$$(31535635 \text{ seconds} \div 60 \text{ seconds}) \div 60 \text{ minutes} = 8759.898611 \text{ hours} \tag{5.5}$$

According to Espressif [65,128], the deep sleep current consumption of an ESP32-H2 is ~ 7 uA, or $7 \cdot 0.001 = 0.007$ mA. The energy consumption of the SED during deep sleep in one year is:

$$0.007 \text{ mA} \cdot 8759.898611 \text{ hours} = 61.31929028 \text{ mAh} \tag{5.6}$$

[7] I learned how to calculate the energy consumption in mAh, given both the duration and the current, from an answer in the Electrical Engineering Stack Exchange forum [132].

The energy consumption of the SED when secured under AES-CCM during the course of a single year is:

$$3.167558896 + 61.31929028 = 64.48684918 \text{ mAh} \tag{5.7}$$

5.3.2.3 Yearly Energy Usage Under ASCON-128a

In Figure 5.5, the average energy consumption of the SED during wakeup, when secured under ASCON-128a, was 32.494022857083735 mA. The SED is on wakeup for 365 seconds in a single year, which Equation 5.1 states is equivalent to 0.1013888889 hours. When using ASCON-128a, the energy consumption of the SED on wakeup is:

$$32.494022857083735 \text{ mA} \cdot 0.1013888889 \text{ hours} = 3.294532873 \text{ mAh} \tag{5.8}$$

Figure 5.3.2.2 showed that the energy consumption of a SED on deep sleep in one year is 61.31929028 mAh. This holds true regardless of the encryption used, since encryption and decryption are never performed during deep sleep.

The yearly energy consumption of a SED secured under ASCON-128a is:

$$3.294532873 + 61.31929028 = 64.61382315 \text{ mAh} \tag{5.9}$$

5.3.2.4 Percentage Differences

When comparing the differences in energy consumption in terms of milliampere (mA), the following percentage increase when ASCON-128a is used, compared to AES-CCM, is:

$$(1 - \frac{32.494022857083735}{31.24167677509521} \cdot 100) = 3.854081372 \tag{5.10}$$

There is an $\approx 3.854\%$ increase in average energy consumption when ASCON-128a is used, compared to AES-CCM.

The percentage difference between ASCON-128a and AES-CCM becomes significantly less when analyzing the energy consumption of the SED over the course of a year:

$$(1 - \frac{64.48684918}{64.61382315} \cdot 100) = 0.1965120833\ \% \tag{5.11}$$

As shown in Equation 5.3.2.4, there is a $\approx 0.197\%$ increase in yearly energy consumption when ASCON-128a is used, compared to when AES-CCM is used instead.

5.3.2.5 Battery Lifetimes

When writing this thesis, the search results generated by Amazon, for the search term "AA Battery mAh", returned results for batteries that had a capacity in the range of $600 - 3700$ mAh [133]. The capacity of the battery returned as

first search result is used: the Amazon Basics rechargeable AA battery, which has a charge of 2400 mAh [134].

When secured by AES-CCM and ASCON-128a, the yearly energy consumption of the SED is 61.31929028 mAh and 64.61382315 mAh, respectively. The battery lifetime of the SED is:

$$\frac{2400 \text{ mAh}}{61.31929028 \text{ mAh per year}} = 39.13939625 \text{ years under AES-CCM} \tag{5.12}$$

$$\frac{2400 \text{ mAh}}{64.61382315 \text{ mAh per year}} = 37.14375474 \text{ years under ASCON-128a} \tag{5.13}$$

The difference in years of battery life between ASCON-128a and AES-CCM is $39.13939625 - 37.14375474 = 1.99564151$ years, which is equivalent to:

$$(1.99564151 \text{ years} \cdot 365 \text{ days}) = 728.4091511 \text{ days} \tag{5.14}$$

and the percentage difference in battery lifetime between ASCON-128a and AES-CCM is:

$$(1 - \frac{37.14375474}{39.13939625}) \cdot 100 = 5.098805044\% \tag{5.15}$$

when a 2400 mAh battery is used to power the SED.

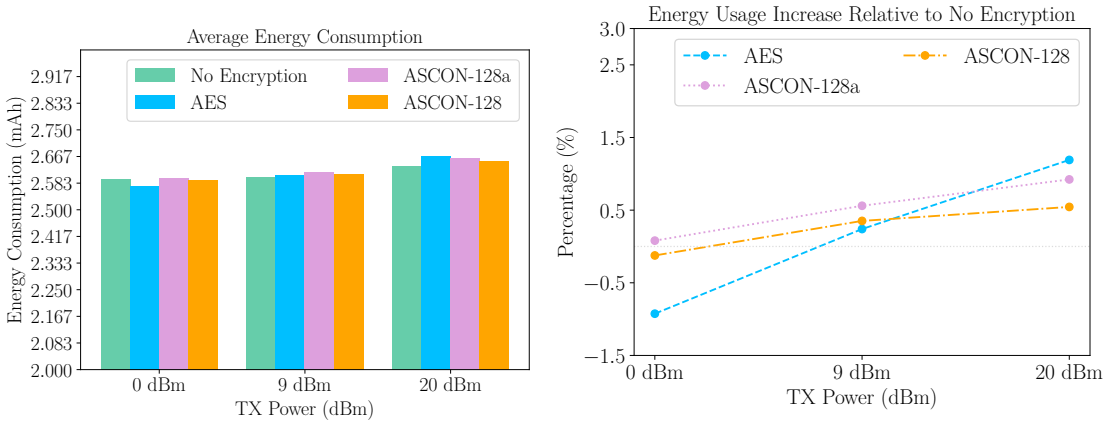
5.3.2.6 Hypothesis

When secured by ASCON-128a, the SED will have a battery lifetime that is $\sim 5.099\%$ less than when AES-CCM is used. This equates to a battery lifetime

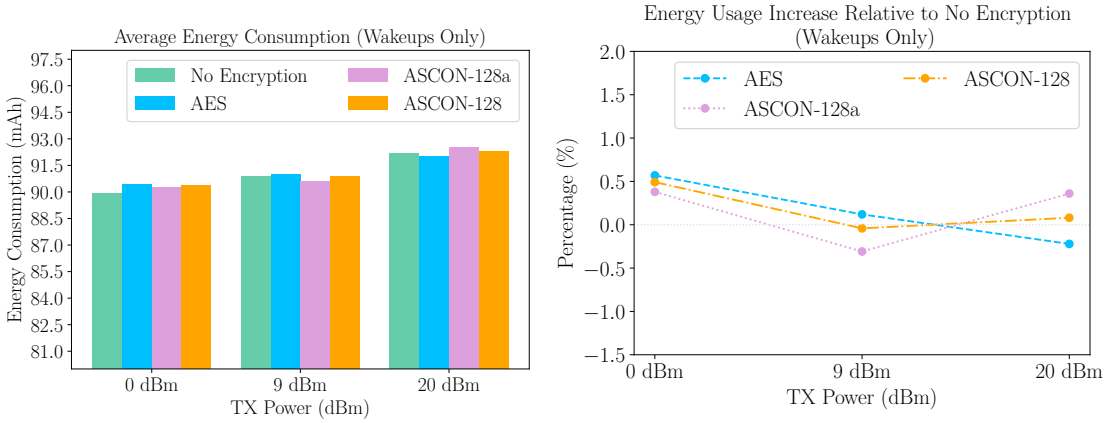
decrease of ~ 1.996 years, or ≈ 729.409 days, when ASCON-128a is utilized, compared to AES-CCM. These differences arise when a 2400 mAh battery is used by the SED.

Although a battery lifetime decrease of ~ 1.996 years, or ~ 729.409 days, is large, such a decrease is not significant when the battery lifetime of the SED will be more than 30 years, regardless of whether ASCON-128a or AES-CCM is used to secure the communications of the SED. This leads to the hypothesis that although there may be an increase in energy consumption and decrease in battery lifetime, when ASCON AEAD is used compared to AES-CCM, such increases in energy consumption and decreases in battery lifetime will not be significant. Thus, there may be no major difference in the energy consumption of the SED when the ASCON is used, compared to AES-CCM. There may only be a slight increase in energy consumption when the ASCON AEAD algorithms are used.

5.3.3 Results



(a) The average energy consumption when the deep sleep current is included in the calculation of the averages.



(b) The average energy consumption when the SED is on wakeup.

Figure 5.6: The results of the Energy Consumption experiments.

The results of the Energy Consumption experiments are shown in Figure 5.6. Every sample current that was recorded by the PPK2 for each experiment was calculated to obtain the average current, in mAh, that are shown in Figure 5.6a. Only the sample currents which equal to or exceeds 9 uA[8] were included in the

calculations in the averages shown in Figure 5.6b to show the average energy consumption when the SED was on wakeup.

The results show that there was no significant difference in energy consumption when the encryption algorithm was varied. As shown in the line graph in Figure 5.6a, when the deep sleep currents were included in the calculations of the average, the increase and decrease in energy consumption would never exceed 2% in either direction. This holds even when the TX power is varied, and the changes become even *less significant* when only the wakeup current is accounted for. The most salient difference shown in the line graph in Figure 5.6b, when the SED has a slightly higher than 0.5% energy usage increase secured under AES-CCM, compared to when plaintext communications were used, when operating at 0 dBm.

The TX power was the factor that made a noticeable difference in the energy usage of the SED. In the bar graphs shown in Figure 5.6, the bars get slightly larger as the TX power increases. This is observed when analyzing the bars for each individual encryption algorithm as the TX power is varied. This is not surprising as the highest peaks in the waveform occur when the device is transmitting the packets that it plans to send. The TX power used SED limits the amount of

[8] Although the deep sleep current of an ESP32-H2 is 7 uA [65,128], in practice, the observed deep sleep current can be as high as 8.84 uA. A threshold of 9 dBm ensured that no deep sleep current samples were added in the calculation of the average energy consumption on wakeup in each experiment.

energy the device can use to transmit packets [135–137].

The insights gathered from the results of the Energy Consumption experiment indicate that it is the TX power, rather than the encryption algorithm, which influenced the energy consumption of the SED. There is no significant difference in the energy consumption of a SED when an encryption algorithm that is different from AES-CCM is used in OpenThread, or when plaintext communications is used, confirming the hypothesis discussed in Section 5.3.2.6.

Chapter 6

Conclusion

6.1 Conclusion

This thesis shows no significant adverse impact in the network performance or energy consumption of ESP32 Thread Devices running a version of OpenThread modified to replace AES-CCM with ASCON AEAD. This result suggests ASCON AEAD is a viable alternative to AES-CCM in any Thread implementation. Indeed, IEEE are currently updating the 802.15.4 standard to include ASCON as an encryption option [16, 17].

The ASCON cipher suite was introduced to address the lightweight encryption problem [18]. NIST has released official versions of ASCON [1, 2] AEAD and hashing that can be mass adopted by the many IoT devices that currently exist

today and that will be created in the future. Since the IEEE plans to add ASCON as an optional alternative to AES-CCM in 802.15.4 [16,17], it may be only a matter of time until commercial devices which operate which under protocols built on top of 802.15.4, such as Thread [33] and Zigbee [138], will be secured under ASCON. Given the current state of the Internet of Things (IoT) today, and where it may head in the future, the contributions of this thesis could potentially be used to help move the world forward towards a more secure future.

6.2 Future Work

This thesis did not investigate how ASCON encryption can be used to secure the communications of SSEDs. There currently exists no published academic research evaluating the network performance and energy consumption of SSEDs. Future work can investigate how ASCON AEAD can be used to secure SSEDs, and the impact on network performance and energy usage when ASCON AEAD is used to secure these devices in place of AES-CCM.

Appendix A

Challenges & Observations

This section describes the challenges encountered when designing and running the Network Performance and Energy Consumption experiments, and how they were addressed. Included are discussions of the interesting observations that were noticed when running these experiments.

A.1 Network Performance Tests

A.1.1 Network Time Synchronization Errors

The delay client and server rely on the OpenThread network time synchronization feature [107–110] to calculate the delay for each sent and acknowledged CoAP packet in the Delay experiments. Under OpenThread network time syn-

chronization, it is responsibility of the network leader for synchronizing the time of all the devices in a Thread network [109].

Since the delay client performed a software restart after each trial, it would not be practical for it to have the responsibility of managing network time synchronization between itself and the delay server. As a result, the delay server was set to be the network leader. This guarantees that the network synchronize time would not be erroneously altered due to software restarts. To maintain this invariant, the delay server was always be powered on *before* the delay client at the beginning of a Delay experiment, where it will form a Thread network where it itself was the network leader. Afterwards, the delay client was powered on and joined the Thread network lead by the delay server.

It was not always guaranteed that the delay server and client could connect into a single Thread network. When this happens, they formed temporary separate network partitions until they could find each other again to merge into a single partition [33, 48] – there were cases in which the delay server and client formed separate partitions that they themselves were the leader of. When the delay client formed its own partition, it would do a software restart and subsequently attempt to attach to the network lead by the delay server. It would repeat this process until it was in the same Thread network as the delay server, then begin the current experimental trial. In the event there was a network time synchronization

problem *during* an experimental trial, the delay client would do a software restart and rerun the trial.

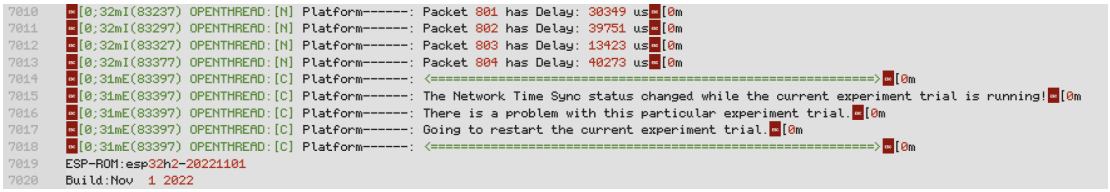


Figure A.1: A network time synchronization error that occurred during a rerun of the Delay experiment when the TX power was set to 0 dBm, and the LibAscon implementation of ASCON-128a was being used. This time synchronization problem occurred during a *rerun* of the Delay experiments. As stated in Section 4.8.1, I did not use the results of the reruns in this thesis.

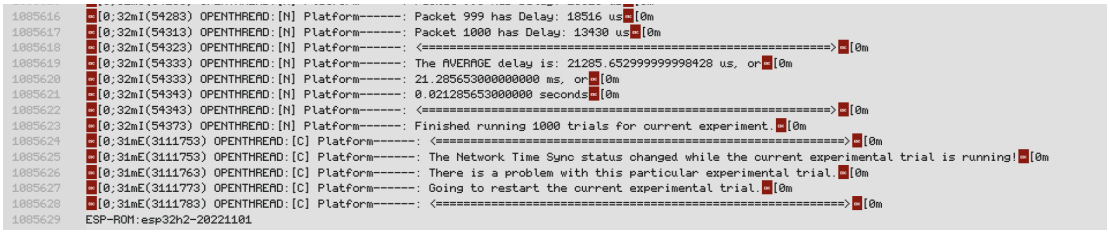


Figure A.2: A network time synchronization error that occurred after the completion of the 1000th trial of the Delay experiment where the devices were running at a TX power of 0 dBm, and the AES-CCM encryption algorithm was used. Since the error occurred *after*[1] the completion of the trial, the results for the trial were still valid.

[1]The Delay experiment for when the encryption algorithm and TX power set to AES-CCM and 0 dBm, respectively, was the only Delay experiment to have a network time synchronization error after the completion of the 1000th trial. Since there was a network time synchronization error, it did do rerun the 1000th trial. However, I did not use the result of the 1001st trial; I only used the averages of the first 1000th trials when creating the graphs shown in Figure 5.1.

A.1.2 Experimental Trial Restarts

There were times in which a trial of a Network Performance experiment would need to be rerun. The reasons a trial rerun would need to occur are described as follows:

Cause 1: When a trial in a Delay experiment encountered a network time synchronization issue, as described in Section A.

Cause 2: When the delay client (in the Delay tests) or FTD (in the Throughput and Packet Loss tests) failed to receive an ACK for a Confirmable request that it sent. For the Delay and Throughput experiments, this occurred in the middle of a trial. Although Non-Confirmable packets primarily used in the Throughput and Packet Loss Observe experiments, timeouts did occur at the beginning or end of a trial, as Confirmable packets that were sent by the Border Router to set up or end an Observe experiment trial failed to be ACKed. The timeout period used in all experiments were set the default values used in OpenThread [139, 140].

Cause 3: As mentioned in Section A, the FTDs in the delay experiments form their own Thread network partitions when they were unable to find each other. This does not only happen in the Delay experiments – this problem occurred in *all* Network Performance experiments.

Whenever any of the issues described above were detected, the FTD (in the Throughput and Packet Loss tests) or delay server (in the Delay tests) performed a software restart and attempted to reattach to the Thread network - and repeated this process until was able to successfully attach. An example of this is seen in Figure A.3. For the Throughput Confirmable in particular, the FTD would do a software restart and attempt network attachment whenever there was a timeout when waiting to receive a ACK for a Confirmable CoAP request. An example of this is displayed in Figure A.4.

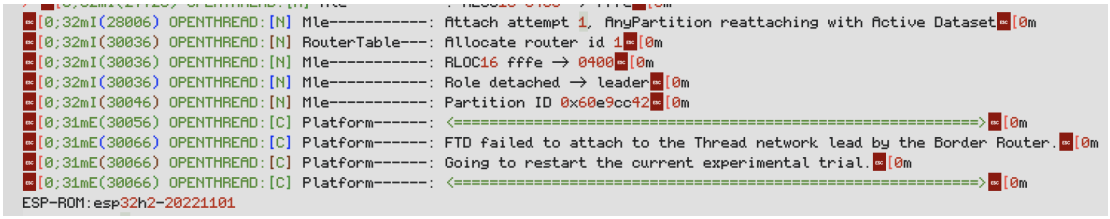


Figure A.3: A trial, which occurred in a rerun[2] of the AES-CCM 20 dBm Throughput Confirmable experiment, in which the FTD did a software restart since it formed its own network partition with itself as the leader.

The Border Router performed software restarts in the Observe experiments when it failed to receive an ACK for any Confirmable CoAP request that it sent[3]. Whenever the Border Router failed to receive an ACK for the Confirmable requests

[2]This trial was run after running the initial Throughput Confirmable experiment. This trial, or any trial in the same experiment, were not used in the results of thesis due to the reasons discussed in Section 4.8.1. No results were obtained from that experiment, as the FTD and Border Router were never able to successfully attach into a single Thread network.

[3]This approach was taken for as I realized doing so could detect whether the Border Router and FTD were in the same network partitions using *fewer* lines of code. In other words, no extra code needed to be written to check whether the Border Router was the leader of a network partition.

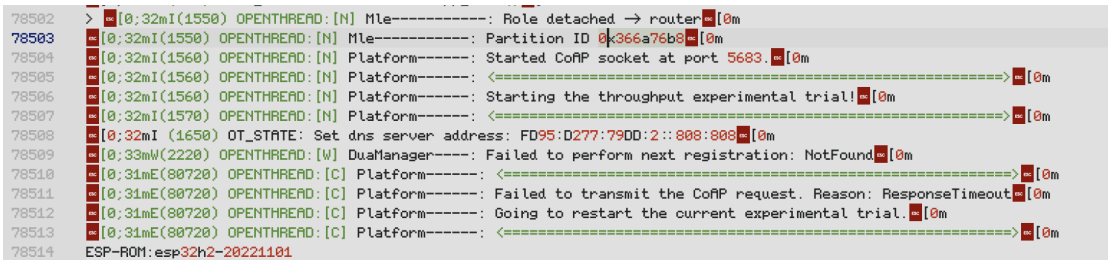


Figure A.4: Software restart performed in the No Encryption 0 dBm Throughput Confirmable experiment when the FTD experienced a timeout after failing to receive an ACK for a Confirmable CoAP request that it sent.

that it sent, the Border Router performed a software restart and attached to the Thread network that the FTD (which was hosting the CoAP Observe endpoint) was in.

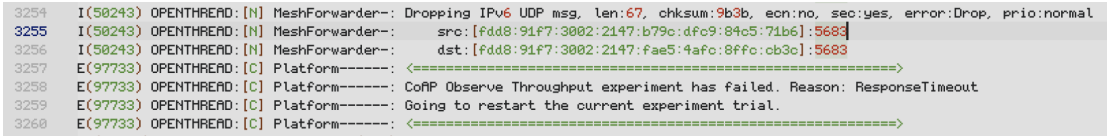


Figure A.5: A trial in the Throughput Observe ASCON-128a 9 dBm experiment in the Border Router did a software restart.

A graph of how often software restarts occurred in the Throughput Confirmable experiments runs that were run to create the results displayed in Figure 5.2 is shown in Figure A.6. Graphs displaying the number of restarts for the Packet Loss Confirmable and Delay experiments were not created, as those experiment runs did not have any. However, I did not create restart graphs for the Throughput and Packet Loss Observe experiments for a different reason: there were too many experimental restarts. There were a significant amount of restarts

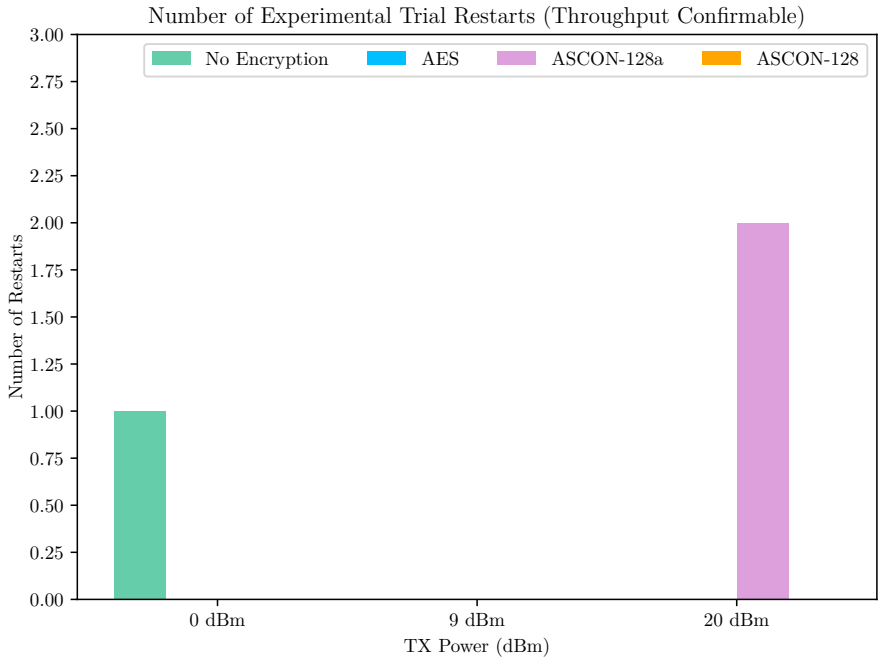


Figure A.6: The number of software restarts done in each Throughput Confirmable experiment due to a error caused by Cause (1), (2), or (3).

which occurred in reruns of the Throughput Confirmable and Delay experiments. However, I did not use the results of these reruns due to the reasons highlighted in Section 4.8.1.

Sometimes, the software restarts were able to resolve the network connectivity issues between the FTD and Border Router, and the trial was able to successfully after one or more subsequent (even sometimes many) software restarts. An example of this is shown in Figure A.7.

[4] Although the `tmux` terminal showed that only two restarts occurred, in actuality, there were 'assertion failures' that occurred, and the FTD had to do a restart that occurred. Even with these issues, the Border Router and FTD were able to eventually connect into a single Thread network and successfully continue running trials, as indicated by the lines `Trial 1 is now complete` and onwards. Further explanation about the odd trial numbering and the "assertion failures" is discussed in Appendix B.

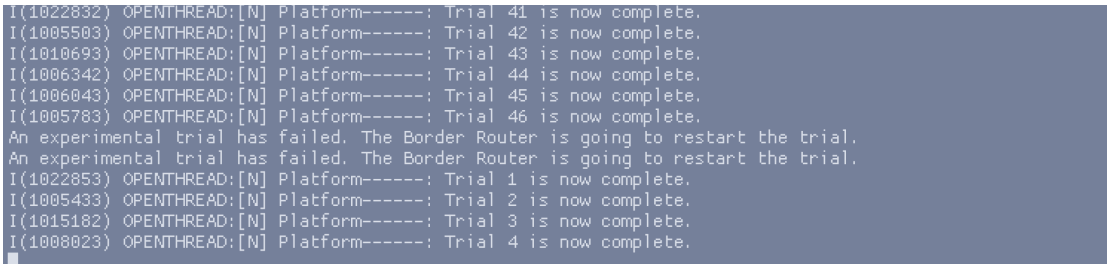


Figure A.7: Restart errors[4] that occurred in the AES-CCM 20 dBm Throughput Observe experiment. These terminal shown is the `tmux` terminal of the Ubuntu Desktop while it is currently running an experiment.

However, it was not always the case that doing software restarts would eventually enable that the FTD and Border Router are connected to the same Thread network. There were runs in which the Thread devices would never be able to attach into a single Thread network. In these events, I had to manually intervene by powering off all the devices and rerunning all the automation scripts. This process of manual intervention was repeated until the devices were able to attach to a single Thread network together and successfully start running experimental trials.

For the Throughput and Packet Loss Observe experiments, there were moments in which there would be n trials would successfully be completed, for $0 < n < 100$. However, there would be a network error in the $n+1$th trial, and the Border Router would keep on performing software restarts and not able to reattach to the network. In such events, I saved the data I had so far, and did

manual intervention to run the remaining $100-n$ trials[5]. These separate sets of data were used in the results for the Observe experiments, shown in Figure 5.2 and 5.3.

A.1.3 Setting TX Power

The commit IDs for the versions of the fork[6] of ESP-IDF [95] that were used in each experiment are listed as follows, in exactly the order in which the experiments were run[7]:

1. Delay: `aa8f5f533667ec3dcf446d2cc5c8e785cc09676b`
2. Throughput Confirmable: `58f5dc26e3614db3a55d6a6f413195f72d2481de`
3. Packet Loss Confirmable: `26f653e472bedbf038a86bf358597242c4095c18`
4. Throughput Observe: `9b33081ad38dae47ee01d6420881e1fcde948b47`
5. Packet Loss Observe: `9b33081ad38dae47ee01d6420881e1fcde948b47`

As each of these experiments were run, the commit of the ESP-IDF fork used in the experiments was update to keep the fork in up to date with the latest changes

[5] I actually ran more than the remaining $100-n$ trials to gather more data, but in thesis, only the results for the *first 100* trials were used.

[6] The commit IDs of the OpenThread fork, were not specified since the fork of the Espressif OpenThread fork is included as a `git` submodule to the ESP-IDF fork.

[7] Reruns were done for the Delay experiment after initially running the Throughput Confirmable experiments. However, the results of the reruns were not used (for reasons described in Section 4.8.1), I did not consider these reruns in the ordering shown in the list below.

that were made by Espressif. The initial plan was rerun the Delay, Throughput, and Packet Loss Confirmable experiments with the same commit ID used in the Throughput and Packet Loss Observe experiments to have results that all use a single, most recent version of ESP-IDF. However, I did not follow through on this approach, as it was infeasible to keep rerunning experiments. Commits of ESP-IDF was *not* changed when running the commits for one particular kind of experiment; the commits were different only between the different types of experiments. All Delay experiments were run at the exactly same commit, and this applies for the Throughput Confirmable, Throughput Observe, Packet Loss Confirmable and Packet Loss Observe experiments as well.

To the best of my knowledge, there was only one difference between the commits which did affect the experiments: with respect to *when* the TX power could be set. Beyond this difference, there were no other noticeable differences between the commits of my ESP-IDF fork that has affected the results of the experiments. I was able to work around this difference and reran the necessary experiments to gather valid data for all Network Performance experiments.

The first set of experiments which were run were the Delay and Throughput Confirmable experiments. Using the OpenThread Radio Configuration API [135–137], the maximum TX power of the ESP32-H2 and ESP Thread Border Route was set before the OpenThread main loop has been launched [62]. This enabled

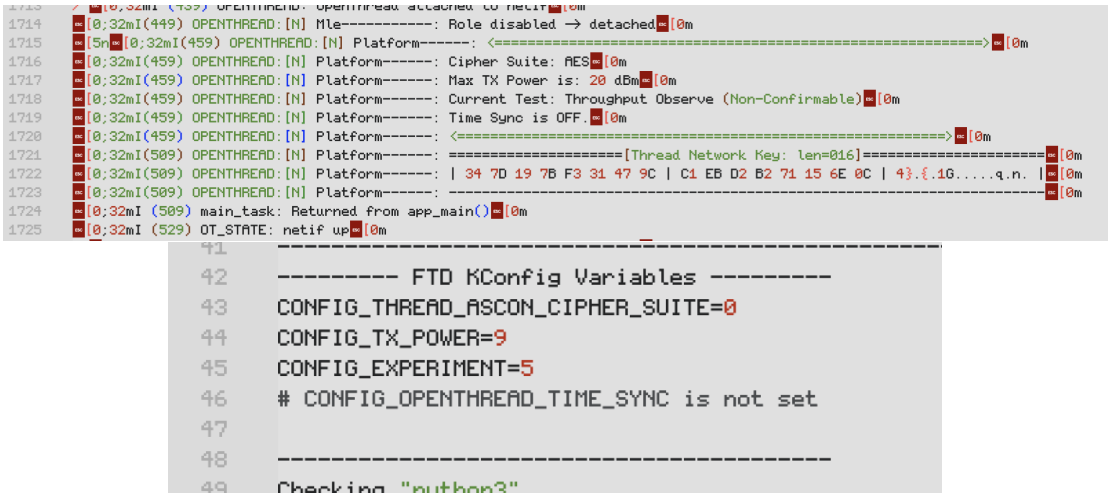


Figure A.8: The TX power of the FTD was at 20 dBm, even though it should have been set to 9 dBm, in this (invalid) run of the AES-CCM 9 dBm Throughput Observe experiment.

the delay client and delay server in the Delay experiments, and the FTD and Border Router in the Throughput Confirmable experiments, to use the desired TX power when attaching to the Thread network and once connected to the Thread network.

To ensure that TX power had been set to the desired value, the OpenThread Radio Configuration API was used to query for the maximum TX power that was set after the API was used to set the TX power. When initially running the Throughput Observe experiments, I observed that the TX powers were not set to the correct value. Examples displaying this issue are shown in Figures A.8 and A.9.

[8]Surprisingly, after the first trial, the TX power was able to successfully set at 9 dBm, and remained at 9 dBm for all subsequent trials.

```
1998  I(33184) OPENTHREAD:[N] Platform------: Started CoAP service at port 5683.
1999  I(33184) OPENTHREAD:[N] Platform------: <==========================================================>
2000  I(33184) OPENTHREAD:[N] Platform------: Starting the Throughput Observe experiment trial!
2001  I(33184) OPENTHREAD:[N] Platform------: <==========================================================>
2002  I(33184) OPENTHREAD:[N] Platform------: ====================[Thread Network Key: len=016]=====================
2003  I(33194) OPENTHREAD:[N] Platform------: | 34 7D 19 7B F3 31 47 9C | C1 EB D2 B2 71 15 6E 0C | 4}.{.1G.....q.n. |
2004  I(33194) OPENTHREAD:[N] Platform------: ----------------------------------------------------------------------
2005  I(33194) OPENTHREAD:[N] Platform------: <==========================================================>
2006  I(33194) OPENTHREAD:[N] Platform------: Cipher Suite: AES
2007  I(33194) OPENTHREAD:[N] Platform------: Max TX Power is currently: 20 dBm.
2008  I(33204) OPENTHREAD:[N] Platform------: <==========================================================>
2009  W(33214) OPENTHREAD:[W] Mle-----------: Failed to process UDP: Duplicated
```

```
61    idf.py build
62    -------Border Router KConfig Variables-----------
63    CONFIG_THREAD_ASCON_CIPHER_SUITE=0
64    CONFIG_TX_POWER=9
65    CONFIG_EXPERIMENT=3
66    # CONFIG_AUTO_UPDATE_RCP is not set
67
68    ------------------------------------------------
69    Executing action: fullclean
```

Figure A.9: For the same experiment shown in Figure A.8, the TX power of the Border Router was at 20 dBm[8] when running the first trial, even though it should have been set to 9 dBm.

Debugging the issue lead to the realization that the TX power remained unchanged when it was set before the OpenThread main loop has been called. When I instead set the TX power to when the Border Router and FTD has already attached to a Thread network, then the TX power gets set properly. As a result, this change was made to ensure that the Throughput and Packet Loss Observe experiments were running at the desired TX power. This did not affect the results of the experiments, as the Throughput and Packet Loss Observe experiments do not begin until the two devices have attached into a single network[9].

I was able to ensure that the TX powers have been set to the correct value at each experimental trial by using the OpenThread Radio Configuration API to

[9]This fact applies to the Delay, Throughput, and Packet Loss Confirmable experiments as well.

query the TX power after it has been set. This TX power value is printed to the serial monitor and saved in the logs containing the data for each experiment. This is not only done for the Observe experiments, but for *every* Network Performance experiment as well – even in the Delay and Throughput Confirmable experiments, before this issue even occurred. The logging of the TX power throughout all experimental trials has enabled me to ensure that the results of the experiments are valid.

A.1.4 Calling CoAP Observe Response Handler When Unsubscribing

```
81778   I(1009012) OPENTHREAD:[N] Platform------: Subscription: 0x6986, Temperature: 79°F, Type: NON, Packet Number: 1000
81779   I(1009692) OPENTHREAD:[N] Platform------: <=========================================================>
81780   I(1009692) OPENTHREAD:[N] Platform------: Received: 1000 packets
81781   I(1009692) OPENTHREAD:[N] Platform------: Packets Lost: 0 packets
81782   I(1009692) OPENTHREAD:[N] Platform------: Expected: 1000 packets
81783   I(1009692) OPENTHREAD:[N] Platform------: Packet Loss Ratio: 0.000000000000000
81784   I(1009702) OPENTHREAD:[N] Platform------: <=========================================================>
81785   W(1009702) OPENTHREAD:[W] Platform------: Stopped response handler from attempting to proccess an empty CoAP observe request.
81786   W(1009702) OPENTHREAD:[W] Platform------: Stopped response handler from attempting to process an empty Message Info object.
81787   I(1010132) OPENTHREAD:[N] Platform------: Cancelled subscription 0x6986.
81788   I(1010142) OPENTHREAD:[N] Platform------: Trial 67 is now complete.
81789   I (1010142) wifi:state: run → init (0x0)
```

Figure A.10: Serial monitor output of the response handler returning after receiving the ACK in response to unsubscribe request. Warning print statements were added to indicate that the response handler was doing a premature return at the end of the trial. Shown above is the log of the serial monitor output of the Border Router during the 67th trial of the No Encryption 20 dBm Packet Loss Observe experiment.

When sending a CoAP request using OpenThread, a response handler function must be passed into the call of the `otCoapSendRequest()` function [140].



Figure A.11: The Border Router crash that occurred when debugging the response handler unsubscribe issue. Print statements were temporarily added to print to the serial monitor immediately before the crash occurs.

This applies even when CoAP Observe is used. The subscription request is sent using `otCoapSendRequest()` (with the observe header set to the proper value) and a response handler is passed. All Observe notifications that received are processed by the response handler. However, the response handler is additionally used to receive the final CoAP response from the server after sending a request to unsubscribe. This is problematic when the final response is an ACK.

In the initial implementation of the Border Router for the Throughput and Packet Loss Observe experiments, the Border Router was instructed to assumed that all responses handled by the response handler were notifications, and as a result, it parsed the packet as a CoAP Observe notification. The response handler that ran in the Border Router assumed that all responses were Observe notifications with a token and a payload that stated the simulated temperature of

the ADU. When it processed an ACK sent by the SED in response to request to cancel a subscription, it attempted to parse the ACK as a a notification, resulting in the Border Router crashing, as shown in Figure A.11.

To avoid this issue when the Border Router received a ACK after unsubscribing, a a boolean variable was added to return `true` when the current experimental trial is finished, and `false` otherwise. When the Border Router finished a trial and sent the Observe request to unsubscribe, the boolean variable is then set to `true`. When the value was set to `true`, the response handler immediately returned before processing the packet, prevents such crashes from occurring.

A.1.5 Packet 0 is not Counted in Delay Experiments

```
[0;32mI(26961) OPENTHREAD:[N] Mle-----------: Attach attempt 1, AnyPartition reattaching with Active Dataset[0m
[0;32mI(27761) OPENTHREAD:[N] Mle-----------: RLOC16 fffe -> b801[0m
[0;32mI(27761) OPENTHREAD:[N] Mle-----------: Role detached -> child[0m
[0;32mI(57291) OPENTHREAD:[N] Platform------: <==========================================================>[0m
[0;32mI(57291) OPENTHREAD:[N] Platform------: Started CoAP socket at port 5683.[0m
[0;32mI(57311) OPENTHREAD:[N] Platform------: The Time Sync Period is 30 seconds.[0m
[0;32mI(57311) OPENTHREAD:[N] Platform------: <==========================================================>[0m
[0;32mI(57401) OPENTHREAD:[N] Platform------: Packet 0 has Delay: 68940 us (going to skip)[0m
[0;32mI(57431) OPENTHREAD:[N] Platform------: Packet 1 has Delay: 18107 us[0m
[0;32mI(57481) OPENTHREAD:[N] Platform------: Packet 2 has Delay: 30173 us[0m
[0;32mI(57541) OPENTHREAD:[N] Platform------: Packet 3 has Delay: 30429 us[0m
[0;32mI(57581) OPENTHREAD:[N] Platform------: Packet 4 has Delay: 30439 us[0m
[0;32mI(57621) OPENTHREAD:[N] Platform------: Packet 5 has Delay: 20329 us[0m
```

Figure A.12: A portion of the log for the first trial of the delay experiment when the cipher used was AES-CCM and the TX power was set to 20 dBm. The 0th delay is an outlier relative to the few other delays shown in this portion of the log. The 0th delay having a significantly larger delay was a trend observed in every trial, regardless of the independent variables.

Each experiment consisted of 1000 trials. In each trial, the delay client sent

a total of 1000 Confirmable CoAP requests to the delay server. The network synchronized timestamp is given in microseconds [110]. The delay client never used the delay in the 0th request in each trial when calculating the average delay. The first delay that is calculated (when $i = 0$) was always an *outlier*. This observation was seen in all delay experiments that were conducted, an example of which is shown in Figure A.12. The average delay was calculated for packets with sequence number in the range $1 \leq i \leq 1000$, which gave a total of 1000 delays calculating the average of in each trial.

A.1.6 Malformed and Time Synchronized Packets

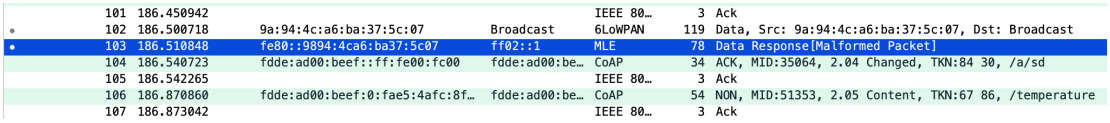


Figure A.13: A malformed MLE packet sent during the AES 20 dBm Throughput Observe experiment.

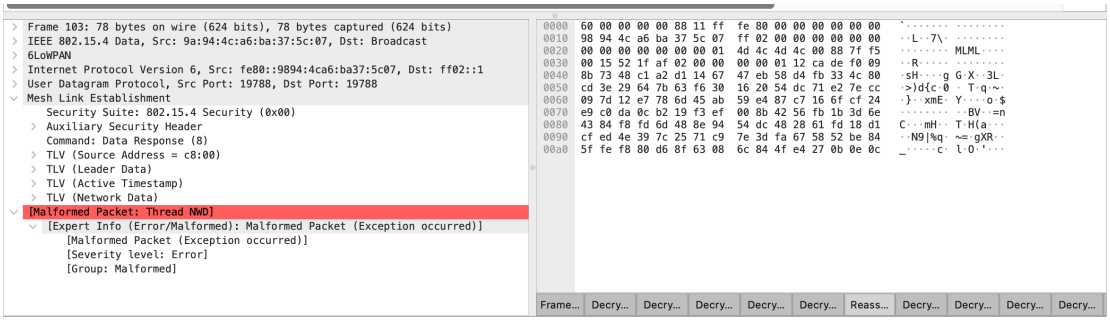


Figure A.14: Details of the malformed packet shown in Figure A.13.

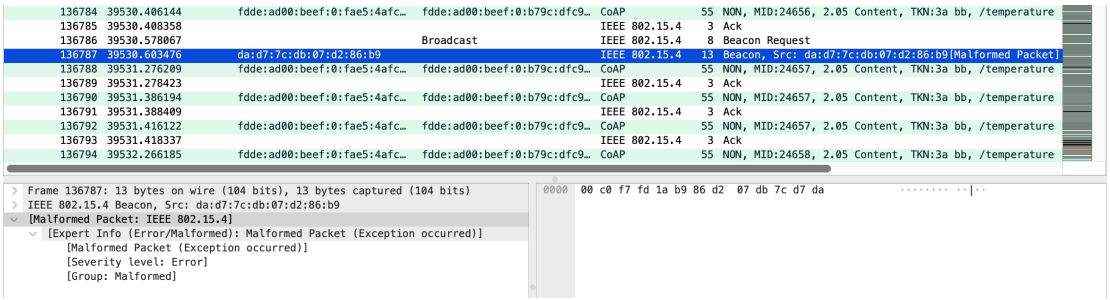


Figure A.15: A malformed 802.15.4 Beacon Response frame seen during the AES 20 dBm Throughput Observe experiment.

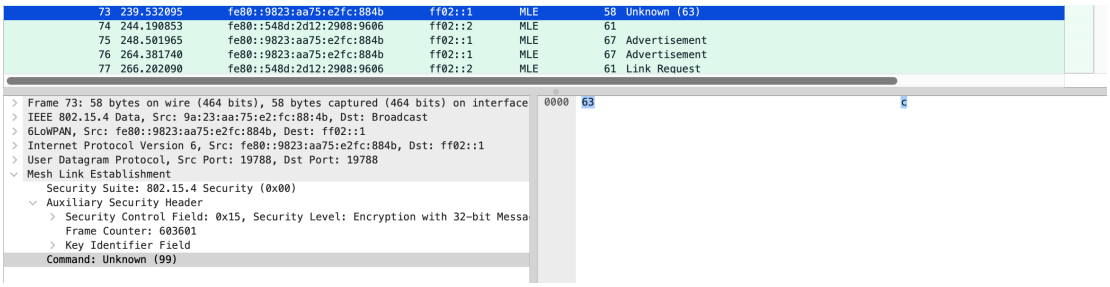


Figure A.16: A packet with an `Unknown` command in the encrypted payload of an MLE packet, seen during the AES-CCM 0 dBm Delay experiment.

When examining packets that were captured by the nRF sniffer in Wireshark, I would observe MLE packets or Beacon Response frames which were malformed – packets which contained MLE or MAC payloads Wireshark could not completely parse. Malformed packets and frames did not appear in the Delay experiments. The fact that the malformed MLE packets were sent by the Border Router, along with how the Border Router was not used in the Delay experiments, entails that the malformed packets and frames too must have been set by the Border Router. These malformed packets were only observed in the Wireshark packets captures

of the experiment in which the AES-CCM encryption algorithm was used, and not when the ASCON-128a, ASCON-128, or no encryption algorithm was utilied, as Wireshark is only able to properly parse AES-CCM packets. Even with these malformed packets, there were no issues in the communication between the Border Router and FTD. As a result, these malformed packets do not affect the validity experiments in any way.

Int the Delay experiments, packets whose payloads were labelled with `Unknown` commands were seen in the packet captures of the AES-CCM Delay experiments. Given that these packets were only seen in the Delay experiments, such MLE packets must be sent as a part of time synchronization feature in OpenThread. This can be inferred since the Delay experiment is the only experiment in which time synchronization was used. The fact that the commands were labelled as `Unknown` further implies this conclusion, as the time synchronization feature is not an official a part of the Thread specification [108]. Since time synchronized packets are not officially part of Thread, it may be that Wireshark may not know how to properly decrypt MLE payloads involved in network time synchronization.

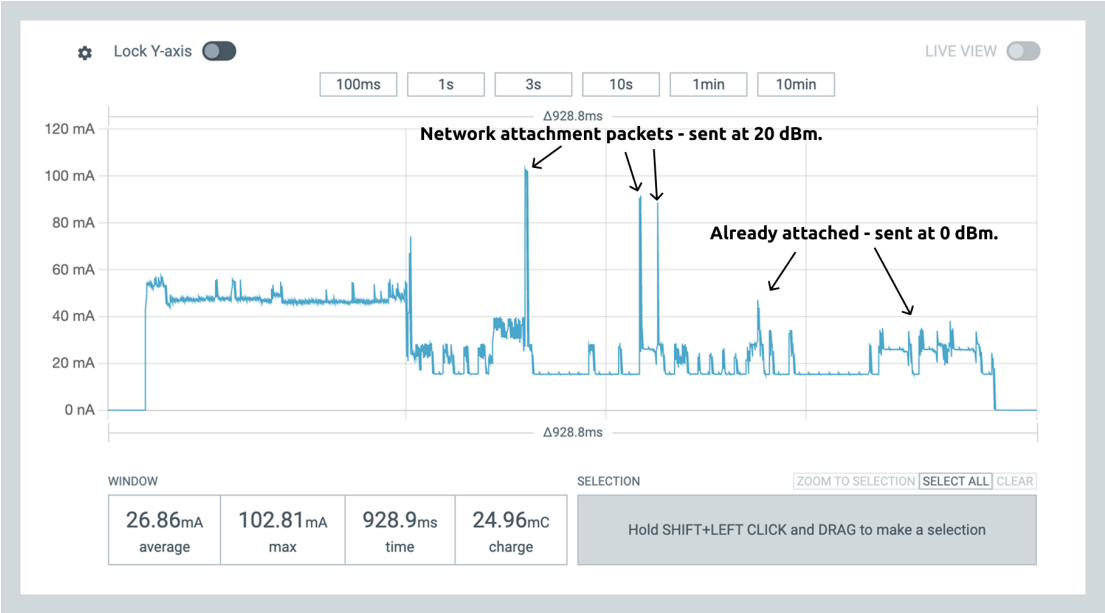


Figure A.17: The waveform diagram of SED (3) during the ASCON-128a 0 dBm experiment. The SED operates at 20 dBm when attaching, then at 0 dBm once it has attached to the network lead by the Border Router.

A.2 Energy Consumption Tests

A.2.1 Setting TX Power

As mentioned in Section 4.8.2, the ESP-IDF repository was set to the same commit ID used in the Throughput and Packet Loss Observe experiments. As a result, the same TX power problem that occurred in the network performance experimsnts (described in Section A), additionally occurred in the Energy Consumption experiments. Thus, the TX power of the SEDs could only be set after they have attached to Thread network lead by the Border Router. The SEDs used

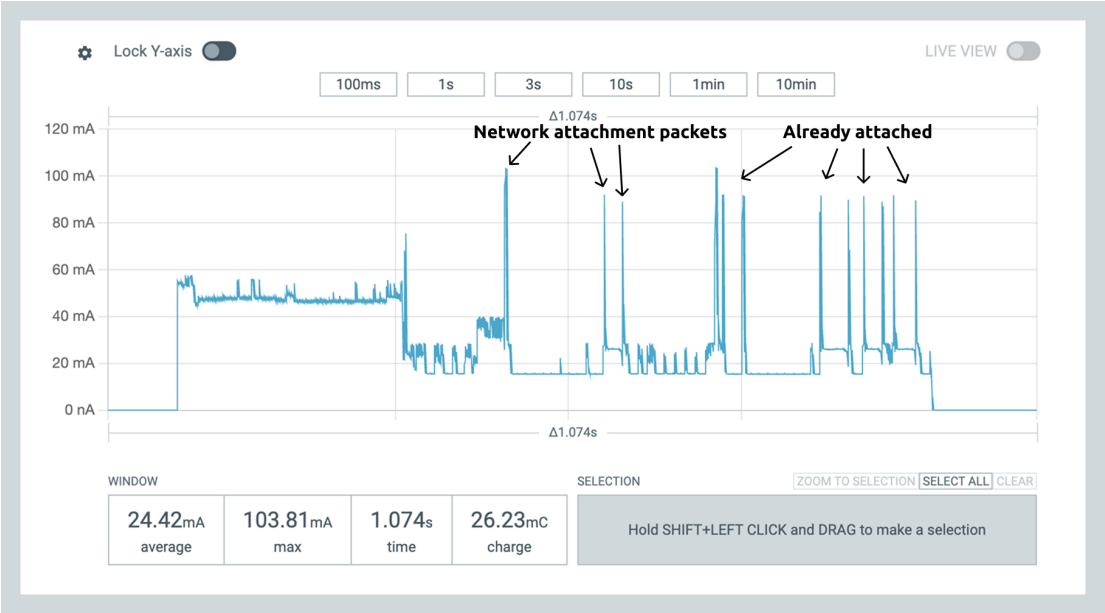


Figure A.18: The waveform diagram of SED (3) during the ASCON-128a 20 dBm experiment. Unlike the waveform shown in Figure A.17, the device always transmitted at 20 dBm.

a default TX power of 20 dBm to transmit packets to the Border Router when attaching to the Thread network. Once attached, the TX powers of the SEDs were set to the proper TX power that is needed for the current experiment.

This does affect the results of the Energy Consumption experiments, since SEDs used a default TX power of 20 dBm to transmit packets to the Border Router when attaching to the Thread network. Once attached, the TX powers of the SEDs were set to the proper TX power that is needed for the current experiment. To analyze the extent to which this affects[10] the energy of the SEDs

[10]This does affect the Border Router as well in the manner described in Section A. However, the TX power set on Border Router does not affect the experiment, as by the SEDs are powered on, the Border Router has set up a Thread network with itself as the leader and has already set

over a given wakeup, we need to refer to the waveform diagram and packet capture shown in Figure 5.4. In the Figure, the packets sent by the SED in order to attach to the network lead by the Border Router corresponding to packet numbers 371, 373, and 376. The first packet that is sent by the SED once it attached to the Thread network was an event packet, which was given packet number 377 by Wireshark.

There were only 3 packets transmitted by the SED to attach to the network lead by the Border Router. Therefore, there should only be 3 peaks in each wakeup cycle which would be transmitted with a TX power of 20 dBm – every other packet will be transmitted with the TX power specified in the given experiment. This trend can be observed in the waveform diagrams given by the `ppk2` data in each of the experiments. This exact trend when comparing the waveforms shown in Figures A.17 and A.18.

Since the SEDs do not transmit at the desired TX power until *after* network attachment, the results shown in Figure 5.6 displays a *less pronounced* difference in energy consumption when the TX power was varied, than what would otherwise be seen if the SED was operating the same TX power throughout the entire wakeup. This does not affect the validity of the Energy Consumption experiments. It further highlights the conclusion that the TX power influences the energy consumption of the SEDs, rather not the encryption algorithm used (or whether none

its TX power accordingly.

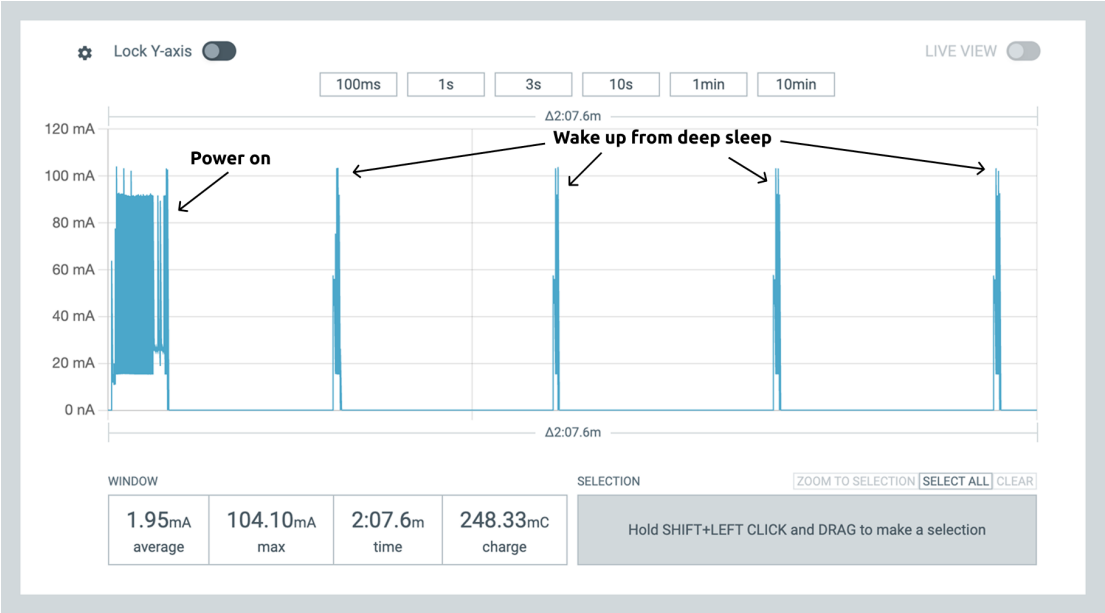
is used at all). Whether the SED was using ASCON AEAD, AES-CCM, or no encryption algorithm at all, was *not* a significant factor that affected the overall energy usage of the SED.

A.2.2 More Attachment Packets Sent on Power On

When SED (3) first powered on, sent several Data Request packets before it would get a response from the Border Router. This occurs at the beginning of every experiment. Since the Data Request packets were sent by the SED to attach to the Thread network, the SED would not have set its TX power yet (for the reason discussed in Section A). On power on, the SED sent all of its Data Request packets with a TX power of 20 dBm. Only once the SED has attached to the network lead by the Border Router, does its TX power get set to the value required by the given experiment. After the initial power on and the SED went to deep sleep, the attachment process for every wakeup afterwards was significantly much shorter, as shown in Figure A.20.

The higher quantity of packets sent before network attachment on power on would increase the average energy consumption of the SED, especially since such Data Request packets would always be sent a TX power of 20 dBm. However, this should not impact the average energy consumption in any significant manner, since this higher quantity of packets were only sent when the SED first powers on. Every wakeup after this initial power-on cycle, the SED only exchanged a few

packets with the Border Router before going back to deep sleep. Thus, this trend is an interesting observation to note, but does not affect the validity of the results collected from the experiments.



184	77.861677			IEEE 802.15.4	3	Ack
185	77.989664	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
186	77.990759			IEEE 802.15.4	3	Ack
187	78.089655	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
188	78.090750			IEEE 802.15.4	3	Ack
189	78.219649	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
190	78.220743			IEEE 802.15.4	3	Ack
191	78.329648	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
192	78.330742			IEEE 802.15.4	3	Ack
193	78.429649	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
194	78.430743			IEEE 802.15.4	3	Ack
195	78.530700	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
196	78.531794			IEEE 802.15.4	3	Ack
197	78.599641	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	70	Child Update Request
198	78.602335			IEEE 802.15.4	3	Ack
199	78.649673	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
200	78.650767			IEEE 802.15.4	3	Ack
201	78.759657	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
202	78.760752			IEEE 802.15.4	3	Ack
203	78.879651	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
204	78.880745			IEEE 802.15.4	3	Ack
205	78.989654	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
206	78.990748			IEEE 802.15.4	3	Ack
207	79.080567	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
208	79.081661			IEEE 802.15.4	3	Ack
209	79.200454	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
210	79.201548			IEEE 802.15.4	3	Ack
211	79.310562	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
212	79.311657			IEEE 802.15.4	3	Ack
213	79.420530	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
214	79.421625			IEEE 802.15.4	3	Ack
215	79.539655	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
216	79.540749			IEEE 802.15.4	3	Ack
217	79.581481	fe80::d8d7:7cdb:7d2:86b9	ff02::1	MLE	67	Advertisement
218	80.329593	fe80::8433:2029:c219:cbd0	ff02::2	MLE	61	Parent Request
219	80.531463	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	115	Parent Response

Figure A.19: The top figure displays the waveforms generated by SED (3) during the first two minutes of the AES-CCM 20 dBm experiment. The bottom figure shows Wirehark packet capture of the higher quantity of back-to-back packets initially SED (3) at the start of the experiment, resulting in the increased energy consumption when the device first powers on.



Figure A.20: The waveforms of SED (3) when first powering on, during the AES-CCM 20 dBm and 0 dBm experiments are shown at the top and bottom, respectively.

A.2.3 When a SED is on wakeup for Longer than 30 Seconds

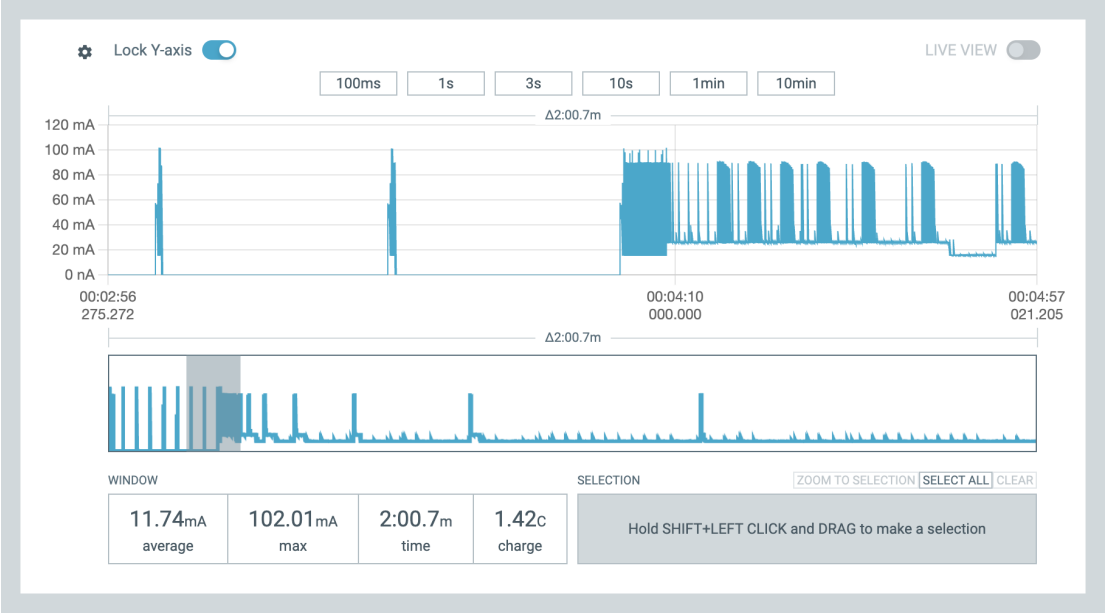


Figure A.21: SED (3) did not go to deep sleep when it had network connection issues in the initial run of the ASCON-128 20 dBm experiment. The minimap is shown to highlight that SED (3) never went into deep sleep. Furthermore, timestamps are shown to note that SED (3) had a network connection issue 4 minutes into the experiment.

Since 30 seconds will simulate one day's worth of smart home network activity in the Energy Consumption experiments, in which each SED will wake up exactly once every day. As a result, the SEDs must wake up once every 30 seconds in each experiment. In the driver code for the SEDs [97], they were instructed *not* to enter deep sleep until they have attached to the Thread network lead by the Border Router, and have sent and receive ACKs for the CoAP requests that the

need to send.

If a SED was unable able to attach to the Thread network lead by the Border Router, or fail to receive ACKs for the packets it has sent, then the SED would not enter deep sleep. In each respective case, it would keep attempting to send packets until it could attach to the Thread network or receive ACKs for the packets it has sent. Thus, there were events in which the SED did eventually attach to the Thread network lead by the SED and receive ACKs for the packets it has sent, and subsequently enter into deep sleep, but take wakeup cycle took longer 30 seconds. The occurrence of such an event was problematic, as every SED should only wake up once every 30 seconds.

To detect when such an issue has occurred, print statements were added to the driver code of the SEDs to print out an error message to UART stating that the SED was awake for more than 30 seconds. When this occurs, the error message is printed out and the SEDs remain awake, not entering deep sleep. This detection mechanism was used to detect this problem on SEDs (4) and (5). However, it was not possible to check UART output of SED (3), as it is power was supplied by the PPK2 and the SED itself was not connected to USB. Since the SED was instructed not to sleep when this issue occurred, I was able to check for the occurrence of this problem by occasionally checking the live energy usage monitoring of the PPK2 while an experiment was running. Whenever this issue was detected, I manually

intervened to stop and rerun the current experiment.

The event in which a SED was on deep sleep for more than 30 seconds only occurred *once* when running the Energy Consumption experiments. The problem occurred when initially running the LibAscon 128 20 dBm experiment. There was a moment in which SED (3) had a network connection issue, and as a result, it was awake for more than 30 seconds. Since the problem has occurred, SED (3) did not go into deep sleep, as shown in Figure A.21. I manually intervened to stop and rerun the ASCON-128 20 dBm experiment. The results of the reran experiments was used in the results for this thesis and *not* the experiment in which the issue, shown in Figure A.21, had occurred.

A.2.4 Malformed Packets

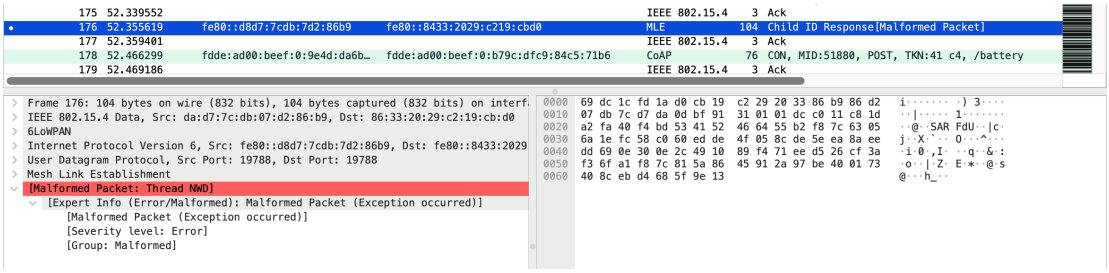


Figure A.22: A malformed Child ID response packet sent by the Border Router during the AES-CCM 0 dBm experiment.

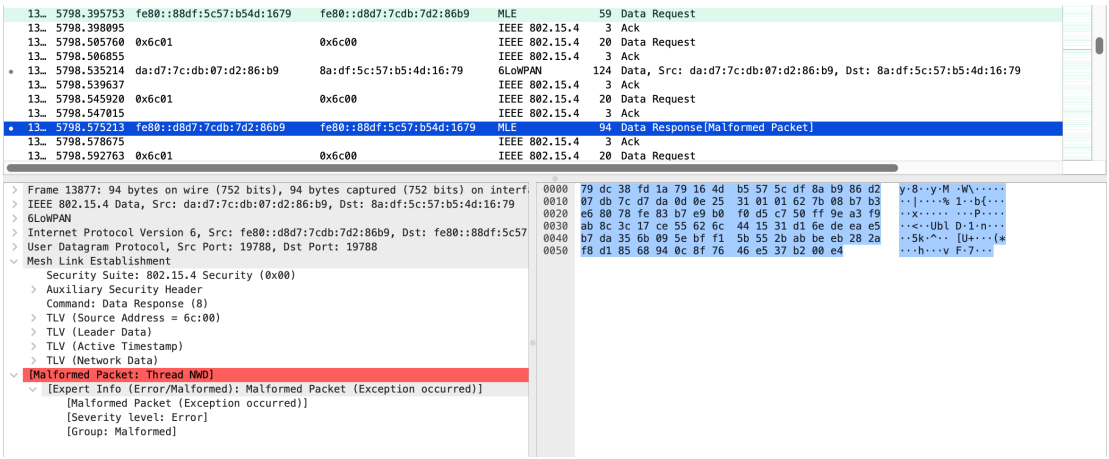


Figure A.23: A malformed Data Response packet sent by the Border Router during the AES-CCM 9 dBm experiment.

Similar to the network performance experiment (as described in Section A), malformed packets were sent by the Border Router in the energy consumption experiments. The SEDs were able to process the malformed MLE packets without any issues. Thus, the malformed packets did not have any noticeable or significant effect on the Energy Consumption experiments. Interestingly, there were no malformed Beacon Response packets that were in-flight observed during the AES-CCM experiments. However, malformed MLE Child ID Response packets were seen by Wireshark during these experiments.

Appendix B

Assertion Failures in Observe Experiments

To ensure that each trial Throughput and Packet Loss Observe experiments were running as intended, mechanisms were added to the Border Router and FTD to ensure this is the case. When this is not the case, these mechanisms will result in warnings and errors to the serial monitor outputs of the respective devices. I manually intervened to stop and rerun an experiment as described in Section A when seeing these warning and error messages.

When the Border Router has established a subscription with the FTD, and up until it cancels its subscription, the only CoAP packets that it should receive were notifications of the simulated temperature of the ADU. This invariant was

enforced during the life of the subscription between the Border Router and FTD by the following assertion function, written the source code of the Border Router [99]:

```
void assertNotification(otMessage *aMessage,
                        Subscription *subscription)
{
  uint64_t token = 0;
  memcpy(&token, otCoapMessageGetToken(aMessage),
         otCoapMessageGetTokenLength(aMessage));

  assert(token == subscription->token);

  uint16_t payloadLength = getPayloadLength(aMessage);
  assert(payloadLength == sizeof(Fahrenheit));
  return;
  }
```

When any of the assertions fail, ESP-IDF by default printed an `assert failed` error and a software restart occurred. An example of an assertion failure is shown in Figure B.4.

With respect to the FTD, when it has an already established subscription with the Border Router, it should *not* expect to receive a CoAP request from the Border Router with a different subscription token. The Border Router should not need to establish a new CoAP Observe subscription with the FTD. Thus, when receiving a CoAP Observe request with a token that was different from the token it has already given the for a previous subscription, the FTD printed a warning

on the serial monitor[1], the subscription request, as shown in Figure B.3.

When an assertion error occurs on the Border Router, and warnings about receiving a new observe subscription request when a subscription has already been established, were displayed in the serial monitor outputs of the FTD and Border Router, respectively, I manually intervened to stop and rerun the experiments (as I have described in Section A). However, there was a case in which I did not need to do manual intervention: when running the Throughput Observe AES-CCM 20 dBm experiments. Even with the warnings and assertion errors, the devices were both able to eventually attach to a single Thread network and ran successful experimental trials afterwards.

The problems began after the 46th Throughput Observe AES-CCM 20 dBm experimental trial was completed. When beginning the 47th trial, the Border Router was unable to set up a CoAP Observe subscription with the FTD. As a result, the Border Router did a software restart and reattached to the same network as the FTD.

However, even though the Border Router did not receive the ACK for its CoAP Observe request, the FTD received the initial Observe request and created a subscription, but the Border Router did not receive the piggybacked ACK and subsequent notifications containing the (simulated) temperature of the ADU.

[1]I had also set up the FTD to also send an ACK to prevent the Border Router from having to retransmitting the Observe request due to not receiving any acknowledgement. This is because the Observe subscription requests were always sent in a Confirmable packet.

```
57084   I(19952) OPENTHREAD:[N] Platform------: Started CoAP service at port 5683.
57085   I(19952) OPENTHREAD:[N] Platform------: <==========================================================>
57086   I(19952) OPENTHREAD:[N] Platform------: Starting the Throughput Observe experiment trial!
57087   I(19952) OPENTHREAD:[N] Platform------: <==========================================================>
57088   I(19962) OPENTHREAD:[N] Platform------: ====================[Thread Network Key: len=016]====================
57089   I(19962) OPENTHREAD:[N] Platform------: | 34 7D 19 7B F3 31 47 9C | C1 EB D2 B2 71 15 6E 0C | 4}.{.1G.....q.n. |
57090   I(19962) OPENTHREAD:[N] Platform------: ------------------------------------------------------------------
57091   I(19962) OPENTHREAD:[N] Platform------: <==========================================================>
57092   I(19962) OPENTHREAD:[N] Platform------: Cipher Suite: AES
57093   I(19972) OPENTHREAD:[N] Platform------: Max TX Power is currently: 20 dBm.
57094   I(19972) OPENTHREAD:[N] Platform------: <==========================================================>
57095   W(20252) OPENTHREAD:[W] DuaManager----: Failed to perform next registration: NotFound
57096   W(21242) OPENTHREAD:[W] DuaManager----: Failed to perform next registration: NotFound
57097   I (21812) OPENTHREAD: Platform UDP bound to port 53536
57098   W(22242) OPENTHREAD:[W] DuaManager----: Failed to perform next registration: NotFound
57099   I (31362) example_connect: Got IPv6 event: Interface "example_netif_sta" address: fd0c:5883:3ceb:d79f:ce8d:a2ff:fe34:72e0, t
57100   I (31362) example_connect: Got IPv6 event: Interface "example_netif_sta" address: fdde:ad00:beef:cafe:ce8d:a2ff:fe34:72e0, t
57101   E(89612) OPENTHREAD:[C] Platform------: <==========================================================>
57102   E(89612) OPENTHREAD:[C] Platform------: CoAP Observe Throughput experiment has failed. Reason: ResponseTimeout
57103   E(89612) OPENTHREAD:[C] Platform------: Going to restart the current experiment trial.
57104   E(89612) OPENTHREAD:[C] Platform------: <==========================================================>
57105   I (89612) wifi:state: run -> init (0x0)
57106   I (89612) wifi:pm stop, total sleep time: 74172199 us / 86192379 us
```

Figure B.1: The Border Router failed to establish a CoAP Observe subscription with the Border Router when starting the 47th experimental trial.

```
2043   [0;32mI(46740219) OPENTHREAD:[N] Platform------: Subscription started with token 0xeb7a.[0m
2044   [0;32mI(46740239) OPENTHREAD:[N] MeshForwarder-: Dropping IPv6 UDP msg, len:57, chksum:0687, ecn:no, sec:yes, error:NoRoute, p
2045   [0;32mI(46740239) OPENTHREAD:[N] MeshForwarder-:     src:[fdd8:91f7:3002:2147:fae5:4afc:8ffc:cb3c]:5683[0m
2046   [0;32mI(46740239) OPENTHREAD:[N] MeshForwarder-:     dst:[fdd8:91f7:3002:2147:b79c:dfc9:84c5:71b6]:5683[0m
2047   [0;32mI(46741229) OPENTHREAD:[N] MeshForwarder-: Dropping IPv6 UDP msg, len:58, chksum:389c, ecn:no, sec:yes, error:NoRoute, p
2048   [0;32mI(46741229) OPENTHREAD:[N] MeshForwarder-:     src:[fdd8:91f7:3002:2147:fae5:4afc:8ffc:cb3c]:5683[0m
2049   [0;32mI(46741229) OPENTHREAD:[N] MeshForwarder-:     dst:[fdd8:91f7:3002:2147:b79c:dfc9:84c5:71b6]:5683[0m
```

Figure B.2: The FTD established an initial CoAP Observe subscription, but the Border Router did not receive the initial piggybacked ACK.

Not knowing of that the subscription was already established by FTD, the Border Router sent another Observe request. However, since a subscription had already been established, the FTD sent an ACK to the Border Router and printed a warning to the serial monitor.

```
OPENTHREAD:[W] Mle-----------: Failed to process UDP: Duplicated[0m
OPENTHREAD:[W] Platform------: Received subscription request from token 0x1989 when already subscribed.[0m
OPENTHREAD:[N] MeshForwarder-: Dropping IPv6 UDP msg, len:54, chksum:34f4, ecn:no, sec:yes, error:NoRoute, p
OPENTHREAD:[N] MeshForwarder-:     src:[fdd8:91f7:3002:2147:fae5:4afc:8ffc:cb3c]:5683[0m
```

Figure B.3: The warning printed to the serial monitor of the FTD that the Border Router is request to establish a CoAP Observe subscription when one has already been established.

Upon receiving the ACK, there was an assertion failure at the Border Router, since the ACK was not a piggyback CoAP ACK containing the simulated temperature of the ADU.

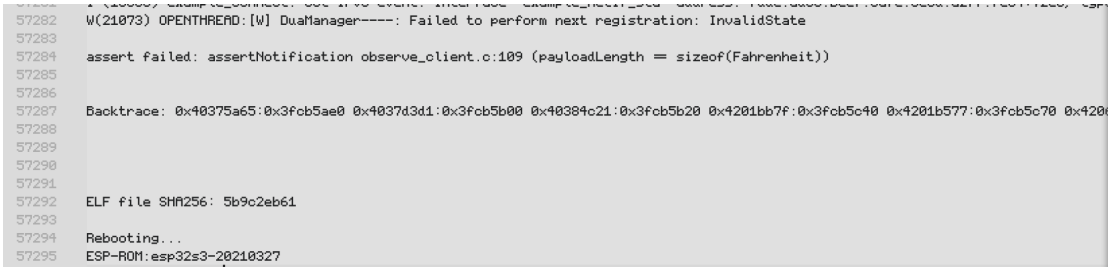


Figure B.4: An assertion failure occurs on the Border Router upon receiving a (non-piggybacked) ACK from the FTD.

As a result, the Border Router did a software restart. This process of the Border Router sending an Observe request, the FTD printing out a warning and sending a (non-piggybacked) ACK, followed by an assertion failure at the Border Router, continues for a total of 464 restarts on the Border Router until the FTD itself has crashed, as shown in Figure B.5.

After crashing, the FTD performed a reboot and set up the CoAP Observe endpoint. It did not retain information about the Observe subscription it previously created before crashing. In the next restart, the Border Router reattached to the Thread network and sent another CoAP Observe subscription request. The FTD received the request and created a new subscription. Afterwards, the FTD and Border Router were able to successfully start and complete the next experimental trial. There were no network connection errors between the Border Router

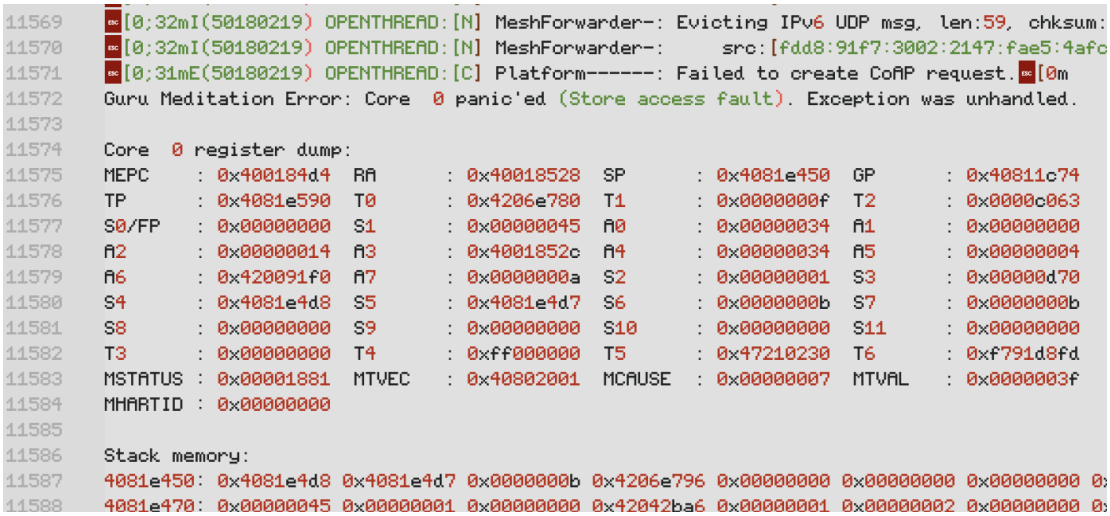


Figure B.5: The FTD crashes after failing to send, and retransmit, too many CoAP Observe notifications to the Border Router.

and FTD after these restarts. As a result, the remaining $100 - 46 = 54$ trials were able to run to completion without any issues.



Figure B.6: After the FTD crash and the Border Router restarts, the two devices were able to attach into a single Thread network and run the remaining 54 experimental trials.

However, due to the repeated restarts, the Border Router had lost information about the number of successful trials it has completed (i.e. it did not remember that it has already successfully completed 46 trials). This is because the `esp_-`

`restart()`[2] function at the end of each trial to invoke a software restart, and the `esp_reset_reason()`[3] function is used to determine whether a software reset occurred, or if the device has just powered on did a restart due to an error. According to the ESP-IDF Programming Guide [143], `esp_reset_reason()` will return the value `ESP_RT_SW` when a restart was invoked by `esp_reset_reason()`.

As a result, the `ESP_RT_SW` was used by the Border Router in the Observe experiments, the FTD in the Confirmable experiments, and the delay client in the Delay experiment, to determine when a new experiment has begun, and thus, to reset the trial count. However, in the particular case of the AES 20 dBm Throughput Observe experiment, the assertion errors caused `esp_reset_reason()` to return a different reason for the reboot, and thus, the Border Router assumed that a new experiment was beginning. As a result, it had reset the number of completed trials from 46 to 0.

Nonetheless, this bug in how the restarts are handled did not affect the validity of the experiment. It only resulted in more trials being run. Rather than manually stopping the experiment after 54 trials were completed, I let the experiment run to 100 trials, in which the automation scripts would end the experiments without the need for any manual intervention. Even though I now had data for 146 trials,

[2] I initially learned about the existence of `esp_restart()` from a forum post in the ESP32 forums [141]

[3] I initially learned about the existence of `esp_reset_reason()` from a forum post in the Arduino forums [142].

I only used 100 trials to create results shown in Figure 5.2: 46 trials before the assertion errors and FTD crash, and the 54 trials after the Border Router and FTD were able to resolve the issues on their own.

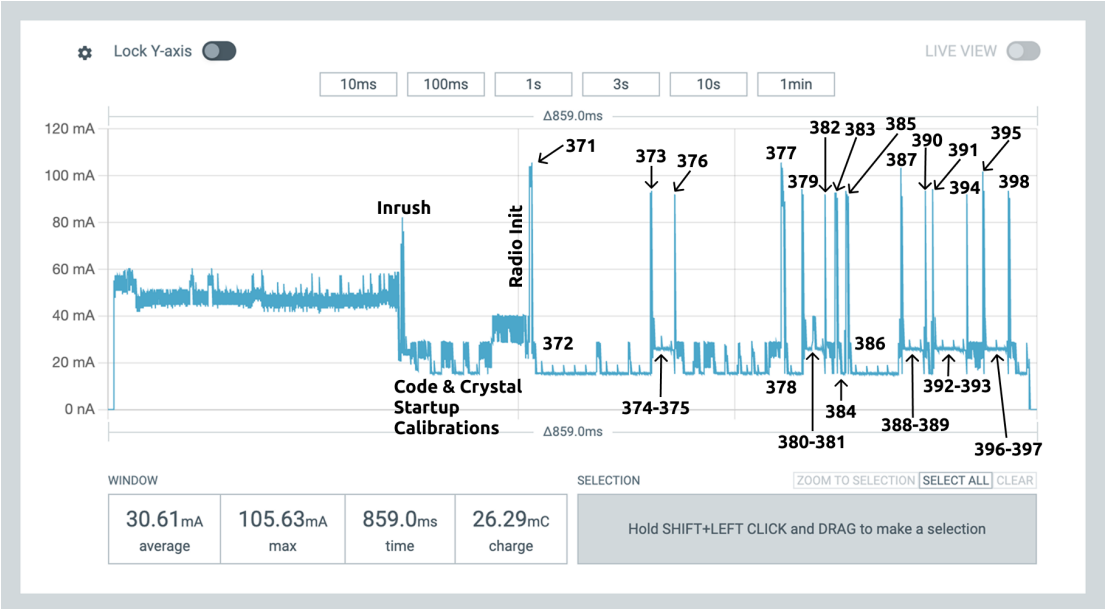
The AES-CCM 20 dBm Observe experiment was the only experiment in which the assertion errors occurred no manual intervention was needed, as the devices were able to recover on their own.

Appendix C

SED Energy Usage Waveforms and Packet Captures

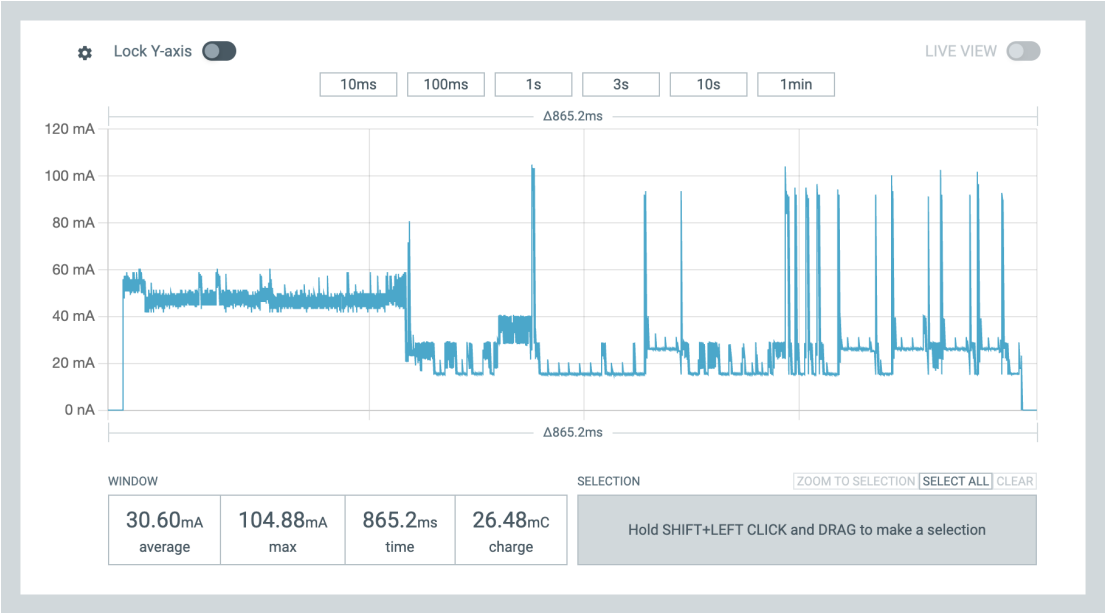
This Appendix chapter contains the waveform diagrams of a SED sending a single Scenario 1 (i.e. battery) and Scenario 2 (i.e. event) packet each to the Thread Border Router in a single wakeup. Each waveform diagram is accompanied by a corresponding packet capture diagram showing every packet sent and received by the SED during the period of time in which energy consumption of the device was measured. Figure 2 from the Thread Battery Operated Devices whitepaper [45] is used as a guide and template when labelling the peaks and valleys for the waveform diagram in Figure C.1. In particular, Figure 2 from the whitepaper was used specifically determine where the inrush, code and crystal

startup calibrations, and radio initialization activities occurred in the waveform shown in Figure C.1. Unlike Figure C.1, the waveform diagrams when the SED was secured under ASCON-128a, ASCON-128, and when not secured under any encryption algorithm, are not labelled, as Wireshark is unable to parse packets that are sent in plaintext, nor decrypt packets sent under the ASCON AEAD algorithms. As a result, these Wireshark packet capture do not give as much useful information as when the packets were encrypted under AES-CCM. Each waveform diagram was recorded at 100k samples per second on the PPK2 and when the SED was operating at a TX power of 20 dBm, and where the waveforms were recorded when the SED sent a single Scenario (1) and Scenario (2) each packet during the single wakeup.



371	246.491384	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	98	Child Update Request
372	246.494078			IEEE 802.15.4	31	Ack
373	246.604168	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
374	246.605263			IEEE 802.15.4	31	Ack
375	246.622999	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	118	Child Update Response
376	246.626332			IEEE 802.15.4	31	Ack
377	246.724873	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	CoAP	102	CON, MID:42724, POST, TKN:87 e6, /event
378	246.727695			IEEE 802.15.4	31	Ack
379	246.744234	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
380	246.745328			IEEE 802.15.4	31	Ack
381	246.763054	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	CoAP	78	ACK, MID:42724, 2.03 Valid, TKN:87 e6, /event
382	246.765108			IEEE 802.15.4	31	Ack
383	246.774220	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	87	Data Request
384	246.776563			IEEE 802.15.4	31	Ack
385	246.784178	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	CoAP	104	CON, MID:42725, POST, TKN:cc 81, /battery
386	246.787065			IEEE 802.15.4	31	Ack
387	246.835105	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
388	246.836199			IEEE 802.15.4	31	Ack
389	246.853046	da:d7:7c:db:07:d2:86:b9	86:33:20:29:c2:19:cb:d0	6LoWPAN	152	Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29
390	246.857469			IEEE 802.15.4	31	Ack
391	246.864337	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
392	246.865431			IEEE 802.15.4	31	Ack
393	246.893069	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	122	Data Response[Malformed Packet]
394	246.896531			IEEE 802.15.4	31	Ack
395	246.910917	0x6c03	0x6c00	IEEE 802.15.4	48	Data Request
396	246.912011			IEEE 802.15.4	31	Ack
397	246.933054	fdde:ad00:beef:0:b79c:dfc9:84c5:71b6	fdde:ad00:beef:0:9e4d:da6b:6b9c:8dd1	CoAP	78	ACK, MID:42725, 2.03 Valid, TKN:cc 81, /battery
398	246.935109			IEEE 802.15.4	31	Ack

Figure C.1: The energy consumption of a SED for a single wakeup when secured under AES-CCM, and operating with the TX power set to 20 dBm.



34	30.219913	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	66	
35	30.222480			IEEE 802.15.4	3	Ack
36	30.325299	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
37	30.326394			IEEE 802.15.4	3	Ack
38	30.355889	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	86	
39	30.359095			IEEE 802.15.4	3	Ack
40	30.456374	0x6c03	0x6c00	IEEE 802.15.4	74	Data, Src: 0x6c03, Dst: 0x6c00
41	30.459196			IEEE 802.15.4	3	Ack
42	30.465335	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
43	30.466429			IEEE 802.15.4	3	Ack
44	30.475304	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	55	
45	30.477518			IEEE 802.15.4	3	Ack
46	30.485539	0x6c03	0x6c00	IEEE 802.15.4	76	Data, Src: 0x6c03, Dst: 0x6c00
47	30.488425			IEEE 802.15.4	3	Ack
48	30.505326	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
49	30.506420			IEEE 802.15.4	3	Ack
50	30.535933	da:d7:7c:db:07:d2:86:b9	86:33:20:29:c2:19:cb:d0	IEEE 802.15.4	124	Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29:c2:19:cb:d0
51	30.540355			IEEE 802.15.4	3	Ack
52	30.555474	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
53	30.556569			IEEE 802.15.4	3	Ack
54	30.585920	da:d7:7c:db:07:d2:86:b9	86:33:20:29:c2:19:cb:d0	IEEE 802.15.4	90	Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29:c2:19:cb:d0
55	30.589254			IEEE 802.15.4	3	Ack
56	30.601083	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
57	30.602178			IEEE 802.15.4	3	Ack
58	30.625933	0x6c00	0x6c03	IEEE 802.15.4	50	Data, Src: 0x6c00, Dst: 0x6c03
59	30.627987			IEEE 802.15.4	3	Ack
60	30.635482	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
61	30.636576			IEEE 802.15.4	3	Ack
62	30.655943	0x6c00	0x6c03	IEEE 802.15.4	50	Data, Src: 0x6c00, Dst: 0x6c03
63	30.657997			IEEE 802.15.4	3	Ack

Figure C.2: The energy consumption of a SED for a single wakeup when packets are sent in plaintext, and when operating with the TX power of 20 dBm.



32	30.219069	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	70	
33	30.221763			IEEE 802.15.4	3	Ack
34	30.313743	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
35	30.314837			IEEE 802.15.4	3	Ack
36	30.332086	fe80::d8d7:7cdb:7d2:86b9	fe80::8433:2029:c219:cbd0	MLE	90	
37	30.335421			IEEE 802.15.4	3	Ack
38	30.432860	0x6c03	0x6c00	IEEE 802.15.4	74	Data, Src: 0x6c03, Dst: 0x6c00
39	30.435683			IEEE 802.15.4	3	Ack
40	30.441836	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
41	30.442931			IEEE 802.15.4	3	Ack
42	30.451868	fe80::8433:2029:c219:cbd0	fe80::d8d7:7cdb:7d2:86b9	MLE	59	
43	30.454211			IEEE 802.15.4	3	Ack
44	30.471818	0x6c03	0x6c00	IEEE 802.15.4	76	Data, Src: 0x6c03, Dst: 0x6c00
45	30.474705			IEEE 802.15.4	3	Ack
46	30.481838	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
47	30.482932			IEEE 802.15.4	3	Ack
48	30.502091	da:d7:7c:db:07:d2:86:b9	86:33:20:29:c2:19:cb:d0	IEEE 802.15.4	124	Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29:c2:19:cb:d0
49	30.506513			IEEE 802.15.4	3	Ack
50	30.511931	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
51	30.513026			IEEE 802.15.4	3	Ack
52	30.532091	da:d7:7c:db:07:d2:86:b9	86:33:20:29:c2:19:cb:d0	IEEE 802.15.4	94	Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29:c2:19:cb:d0
53	30.535553			IEEE 802.15.4	3	Ack
54	30.548366	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
55	30.549461			IEEE 802.15.4	3	Ack
56	30.572100	0x6c00	0x6c03	IEEE 802.15.4	50	Data, Src: 0x6c00, Dst: 0x6c03
57	30.574155			IEEE 802.15.4	3	Ack
58	30.581945	0x6c03	0x6c00	IEEE 802.15.4	20	Data Request
59	30.583039			IEEE 802.15.4	3	Ack
60	30.602095	0x6c00	0x6c03	IEEE 802.15.4	50	Data, Src: 0x6c00, Dst: 0x6c03
61	30.604150			IEEE 802.15.4	3	Ack

Figure C.3: The energy consumption of a SED for a single wakeup when packets are secured under ASCON-128a, and when operating with the TX power of 20 dBm.




```
 32  30.202795   fe80::8433:2029:c219:cbd0   fe80::d8d7:7cdb:7d2:86b9   MLE              70
 33  30.205489                               IEEE 802.15.4               3  Ack
 34  30.305979   0x6c03                      0x6c00                     IEEE 802.15.4   20  Data Request
 35  30.307074                               IEEE 802.15.4               3  Ack
 36  30.321584   fe80::d8d7:7cdb:7d2:86b9   fe80::8433:2029:c219:cbd0  MLE              90
 37  30.324918                               IEEE 802.15.4               3  Ack
 38  30.427371   0x6c03                      0x6c00                     IEEE 802.15.4   74  Data, Src: 0x6c03, Dst: 0x6c00
 39  30.430194                               IEEE 802.15.4               3  Ack
 40  30.445986   0x6c03                      0x6c00                     IEEE 802.15.4   20  Data Request
 41  30.447081                               IEEE 802.15.4               3  Ack
 42  30.461586   0x6c00                      0x6c03                     IEEE 802.15.4   50  Data, Src: 0x6c00, Dst: 0x6c03
 43  30.463641                               IEEE 802.15.4               3  Ack
 44  30.476088   fe80::8433:2029:c219:cbd0   fe80::d8d7:7cdb:7d2:86b9   MLE              59
 45  30.478430                               IEEE 802.15.4               3  Ack
 46  30.482324   0x6c03                      0x6c00                     IEEE 802.15.4   76  Data, Src: 0x6c03, Dst: 0x6c00
 47  30.485210                               IEEE 802.15.4               3  Ack
 48  30.547583   0x6c03                      0x6c00                     IEEE 802.15.4   20  Data Request
 49  30.548677                               IEEE 802.15.4               3  Ack
 50  30.571591   da:d7:7c:db:07:d2:86:b9     86:33:20:29:c2:19:cb:d0    IEEE 802.15.4  124  Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29:c2:19:cb:d0
 51  30.576014                               IEEE 802.15.4               3  Ack
 52  30.586086   0x6c03                      0x6c00                     IEEE 802.15.4   20  Data Request
 53  30.587180                               IEEE 802.15.4               3  Ack
 54  30.611588   da:d7:7c:db:07:d2:86:b9     86:33:20:29:c2:19:cb:d0    IEEE 802.15.4   94  Data, Src: da:d7:7c:db:07:d2:86:b9, Dst: 86:33:20:29:c2:19:cb:d0
 55  30.615051                               IEEE 802.15.4               3  Ack
 56  30.627990   0x6c03                      0x6c00                     IEEE 802.15.4   20  Data Request
 57  30.629084                               IEEE 802.15.4               3  Ack
 58  30.651598   0x6c00                      0x6c03                     IEEE 802.15.4   50  Data, Src: 0x6c00, Dst: 0x6c03
 59  30.653652                               IEEE 802.15.4               3  Ack
```

Figure C.4: The energy consumption of a SED for a single wakeup when packets are secured under ASCON-128, and when operating with the TX power of 20 dBm.

Bibliography

[1] M. Sonmez Turan, "Ascon-Based Lightweight Cryptography Standards for Constrained Devices," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 800-232, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-232.pdf

[2] National Institute of Standards and Technology, "NIST Finalizes 'Lightweight Cryptography' Standard to Protect Small Devices," *NIST*, Aug. 2025, last Modified: 2025-08-18T10:04-04:00. [Online]. Available: https://www.nist.gov/news-events/news/2025/08/nist-finalizes-lightweight-cryptography-standard-protect-small-devices

[3] "TheMatjaz/LibAscon," Mar. 2024, original-date: 2020-05-21T17:25:26Z. [Online]. Available: https://github.com/TheMatjaz/LibAscon

[4] "ascon/ascon-c," Aug. 2025, original-date: 2019-04-23T10:23:58Z. [Online]. Available: https://github.com/ascon/ascon-c

[5] V. Sharma, I. You, K. Andersson, F. Palmieri, M. H. Rehmani, and J. Lim, "Security, Privacy and Trust for Smart Mobile- Internet of Things (M-IoT): A Survey," *IEEE Access*, vol. 8, pp. 167 123–167 163, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9187908/

[6] V. Hassija, V. Chamola, V. Saxena, D. Jain, P. Goyal, and B. Sikdar, "A Survey on IoT Security: Application Areas, Security Threats, and Solution Architectures," *IEEE Access*, vol. 7, pp. 82 721–82 743, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8742551/

[7] M. b. Mohamad Noor and W. H. Hassan, "Current research on Internet of Things (IoT) security: A survey," *Computer Networks*, vol. 148, pp. 283–294, Jan. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128618307035

[8] F. Meneghello, M. Calore, D. Zucchetto, M. Polese, and A. Zanella, "IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8182–8201, Oct. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8796409/

[9] M. N. Khan, A. Rao, and S. Camtepe, "Lightweight Cryptographic Protocols for IoT-Constrained Devices: A Survey," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4132–4156, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9205259/

[10] V. A. Thakor, M. A. Razzaque, and M. R. A. Khandaker, "Lightweight Cryptography Algorithms for Resource-Constrained IoT Devices: A Review, Comparison and Research Opportunities," *IEEE Access*, vol. 9, pp. 28 177–28 193, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9328432/

[11] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "IoT Security: Ongoing Challenges and Research Opportunities," in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*. Matsue, Japan: IEEE, Nov. 2014, pp. 230–234. [Online]. Available: http://ieeexplore.ieee.org/document/6978614/

[12] I. Bhardwaj, A. Kumar, and M. Bansal, "A review on lightweight cryptography algorithms for data security and authentication in IoTs," in *2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)*. solan, India: IEEE, Sep. 2017, pp. 504–509. [Online]. Available: http://ieeexplore.ieee.org/document/8269731/

[13] J. M. Carracedo, M. Milliken, P. K. Chouhan, B. Scotney, Z. Lin, A. Sajjad, and M. Shackleton, "Cryptography for Security in IoT," in *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*. Valencia: IEEE, Oct. 2018, pp. 23–30. [Online]. Available: https://ieeexplore.ieee.org/document/8554634/

[14] M. Sonmez Turan, K. McKay, D. Chang, L. E. Bassham, J. Kang, N. D. Waller, J. M. Kelsey, and D. Hong, "Status report on the final round of the NIST lightweight cryptography standardization process," National Institute of Standards and Technology (U.S.), Gaithersburg, MD, Tech. Rep. NIST IR 8454, Jun. 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2023/NIST.IR.8454.pdf

[15] IEEE Standards Committee, "Amendment to IEEE Standard 802.15.4-2020." [Online]. Available: https://mentor.ieee.org/802.15/dcn/24/15-24-0267-02-cryp-par-for-tg4ae-ascon-for-802-15-4.pdf

[16] "P802.15.4ae: IEEE Standard for Low-Rate Wireless Networks Amendment: Ascon cryptographic algorithms." [Online]. Available: https://standards.ieee.org/ieee/802.15.4ae/11836/

[17] IEEE Standards Committee, "P802.15.4ae: Amendment to IEEE Standard 802.15.4-2020." [Online]. Available: https://development.standards.ieee.org/myproject-web/public/view.html#pardetail/11985

[18] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight Authenticated Encryption and Hashing," *Journal of Cryptology*, vol. 34, no. 3, p. 33, Jun. 2021. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9

[19] Y. Watanabe, H. Yamamoto, and H. Yoshida, "Performance Evaluation of NIST LWC Finalists on AVR ATmega and ARM Cortex-M3 Microcontrollers," 2022, publication info: Preprint. [Online]. Available: https://eprint.iacr.org/2022/1071

[20] R. Ankele and R. Ankele, "Software Benchmarking of the 2nd round CAESAR Candidates," 2016, publication info: Preprint. [Online]. Available: https://eprint.iacr.org/2016/740

[21] J. Avery, B. Fraelich, W. Duran, A. Lee, A. Sullivan, Z. Mechalke, M. B. Birrer, S. Dick, and J. Cochran, "Analysis of Practical Application of Lightweight Cryptographic Algorithm ASCON." [Online]. Available: https://csrc.nist.gov/csrc/media/Events/2022/lightweight-cryptography-workshop-2022/documents/papers/analysis-of-practical-application-of-lwc-cryptographic-algorithm-ascon.pdf

[22] W. BenMassaoud, D. M, R. H. Jhaveri, and G. Srivastava, "Securing Internet of Vehicles Protocols using ASCON and GIFT-COFB," in *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*. Florence, Italy: IEEE, Jun. 2023, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/10199329/

[23] M. Nooruddin and D. Valles, "An Advanced IoT Framework for Long Range Connectivity and Secure Data Transmission Leveraging LoRa and ASCON Encryption," in *2023 IEEE World AI IoT Congress (AIIoT)*, Jun. 2023, pp. 0583–0589. [Online]. Available: https://ieeexplore.ieee.org/document/10174401

[24] S. Neidig, J. Hui, and K. Sporre, "Elegantly connecting your smart home network." [Online]. Available: https://www.threadgroup.org/Portals/0/documents/ElegantlyConnectingYourSmartHomeNetworkWhitePaper_4431_1.pdf

[25] Thread Group, "Thread smart home fact sheet." [Online]. Available: https://portal.threadgroup.org/DesktopModules/Inventures_Document/FileDownload.aspx?ContentID=834

[26] Google. OpenThread. [Online]. Available: openthread.io

[27] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," National Institute of Standards and Technology (U.S.), Gaithersburg, MD, Tech. Rep. NIST FIPS 197-upd1, May 2023. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf

[28] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback, "Report on the development of the Advanced Encryption Standard (AES)," *Journal of Research of the National Institute of Standards and Technology*, vol. 106, no. 3, p. 511, May 2001. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/jres/106/3/j63nec.pdf

[29] National Institute of Standards and Technology, "Block Cipher Modes - Block Cipher Techniques | CSRC | CSRC," Jan. 2017. [Online]. Available: https://csrc.nist.gov/Projects/block-cipher-techniques/BCM

[30] M. Dworkin, "Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality," *National Institute of Standards and Technology*, Jul. 2007. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf

[31] IEEE Computer Society, "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs)," *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pp. 1–320, Sep. 2006, conference Name: IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003). [Online]. Available: https://ieeexplore.ieee.org/document/1700009

[32] ——, "IEEE Standard for Low-Rate Wireless Networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, Apr. 2016, conference Name: IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011). [Online]. Available: https://ieeexplore.ieee.org/document/7460875

[33] Robert Alexander, Sorin Aliciuc, Manuel Amutio, Skip Ashton, Kallola Bal, Przemyslaw Bida, and et al., "Thread specification." [Online]. Available: https://www.threadgroup.org/ThreadSpec

[34] D. J. Bernstein, "Crypto competitions: CAESAR submissions," Feb. 2019. [Online]. Available: https://competitions.cr.yp.to/caesar-submissions.html

[35] J. P. Mattsson, G. Selander, S. Paavolainen, F. Karakoç, M. Tiloca, and R. Moskowitz, "Proposals for Standardization of the Ascon Family," *National Institute of Standards and Technology*. [Online]. Available: https://csrc.nist.gov/csrc/media/Events/2023/lightweight-cryptography-workshop-2023/documents/accepted-papers/03-proposals-for-standardization-of-ascon-family.pdf

[36] Martin Schläffer, Florian Mendel, Christoph Dobraunig, and Maria Eichlseder, "ASCON - Implementations." [Online]. Available: https://ascon.iaik.tugraz.at/implementations.html

[37] "ascon/ascon_collection," May 2025, original-date: 2014-04-09T14:08:46Z. [Online]. Available: https://github.com/ascon/ascon_collection

[38] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," Internet Engineering Task Force, Request for Comments RFC 7252, Jun. 2014, num Pages: 112. [Online]. Available: https://datatracker.ietf.org/doc/rfc7252

[39] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," Internet Engineering Task Force, Request for Comments RFC 6347, Jan. 2012, num Pages: 32. [Online]. Available: https://datatracker.ietf.org/doc/rfc6347

[40] Thread Group, "Thread commissioning." [Online]. Available: https://www.threadgroup.org/Portals/0/documents/support/CommissioningWhitePaper_658_2.pdf

[41] K. Hartke, "Observing Resources in the Constrained Application Protocol (CoAP)," Internet Engineering Task Force, Request for Comments RFC 7641, Sep. 2015, num Pages: 30. [Online]. Available: https://datatracker.ietf.org/doc/rfc7641

[42] J. P. Tuohy. What is thread and how will it help your smart home? [Online]. Available: https://www.theverge.com/23165855/thread-smart-home-protocol-matter-apple-google-interview

[43] K. Ingram, "Thread in commercial backgrounder." [Online]. Available: https://www.threadgroup.org/Portals/0/documents/support/ThreadInCommercialBackgrounder_2710_1.pdf

[44] Thread Group, "Thread smart building fact sheet." [Online]. Available: https://portal.threadgroup.org/DesktopModules/Inventures_Document/FileDownload.aspx?ContentID=2495

[45] ——, "Battery operated devices." [Online]. Available: https://www.threadgroup.org/Portals/0/documents/support/BatteryOperatedDevicesWhitePaper_656_2.pdf

[46] ——, "Thread network fundamentals white paper."

[47] ——. What is thread? | OpenThread. [Online]. Available: https://openthread.io/guides/thread-primer

[48] OpenThread. Node roles and types. [Online]. Available: https://openthread.io/guides/thread-primer/node-roles-and-types

[49] Thread Group. Become a member - thread group. [Online]. Available: https://www.threadgroup.org/thread-group

[50] ——. Why certify - thread. [Online]. Available: https://www.threadgroup.org/What-is-Thread/Certification

[51] S. Neidig, J. Hui, T. Sciorilli, G. Kassel, N. Dyck, and M. Borins, "Thread 1.3.0 webinar." [Online]. Available: https://portal.threadgroup.org/DesktopModules/Inventures_Document/FileDownload.aspx?ContentID=4076

[52] openthread/openthread: OpenThread released by google is an open-source implementation of the thread networking protocol. [Online]. Available: https://github.com/openthread/openthread

[53] OpenThread. OpenThread - getting started. [Online]. Available: https://openthread.io/guides

[54] Amazon, "Matter - Alexa Smart Home." [Online]. Available: https://www.amazon.com/b?node=37490568011

[55] Connectivity Standards Alliance, "Matter - connected home over IP," original-date: 2020-03-03T17:05:10Z. [Online]. Available: https://github.com/project-chip/connectedhomeip

[56] ——. Build with matter | smart home device solution. [Online]. Available: https://csa-iot.org/all-solutions/matter/

[57] Qorvo. Matter gets everybody "talking". [Online]. Available: https://www.qorvo.com/design-hub/blog/matter-gets-everybody-talking

[58] Connectivity Standards Alliance. Matter FAQs | frequently asked questions. [Online]. Available: https://csa-iot.org/all-solutions/matter/matter-faq/

[59] Espressif, "1. Introduction." [Online]. Available: https://docs.espressif.com/projects/esp-matter/en/latest/esp32/introduction.html

[60] "espressif/esp-idf," Apr. 2024, original-date: 2016-08-17T10:40:35Z. [Online]. Available: https://github.com/espressif/esp-idf

[61] Espressif, "Get Started - ESP32 - — ESP-IDF Programming Guide v5.2.1 documentation." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/get-started/index.html

[62] ——, "OpenThread - ESP32." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/openthread.html

[63] ——, "ESP32-H2-DevKitM-1." [Online]. Available: https://docs.espressif.com/projects/espressif-esp-dev-kits/en/latest/esp32h2/esp32-h2-devkitm-1/user_guide.html

[64] ——, "ESP32-H2-MINI-1 and ESP32-H2-MINI-1U Datasheet," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-h2-mini-1_mini-1u_datasheet_en.pdf

[65] ——, "ESP32-H2 Datasheet," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-h2_datasheet_en.pdf

[66] ——, "ESP32-C6 Datasheet," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c6_datasheet_en.pdf

[67] ——, "ESP32-H2 Thread/Zigbee & BLE 5 SoC." [Online]. Available: https://www.espressif.com/en/products/socs/esp32-h2

[68] ——, "ESP Thread BR." [Online]. Available: https://docs.espressif.com/projects/esp-thread-br/en/latest/hardware_platforms.html

[69] "espressif/esp-thread-br," Apr. 2024, original-date: 2022-06-15T12:06:21Z. [Online]. Available: https://github.com/espressif/esp-thread-br

[70] OpenThread, "Co-Processor Designs," Sep. 2023. [Online]. Available: https://openthread.io/platforms/co-processor

[71] R. S. Quattlebaum and j. woodyatt, "Spinel Host-Controller Protocol," Internet Engineering Task Force, Internet Draft draft-rquattle-spinel-unified-00, May 2017, num Pages: 78. [Online]. Available: https://datatracker.ietf.org/doc/draft-rquattle-spinel-unified

[72] "esp-idf/components/openthread/src/spinel at master · espressif/esp-idf." [Online]. Available: https://github.com/espressif/esp-idf/tree/master/components/openthread/src/spinel

[73] "esp-idf/examples/openthread/ot_sleepy_device/deep_sleep at master · espressif/esp-idf." [Online]. Available: https://github.com/espressif/esp-idf/tree/master/examples/openthread/ot_sleepy_device/deep_sleep

[74] "esp-idf/examples/openthread/ot_sleepy_device/light_sleep at master · espressif/esp-idf." [Online]. Available: https://github.com/espressif/esp-idf/tree/master/examples/openthread/ot_sleepy_device/light_sleep

[75] Espressif, "Sleep Modes - ESP32 - — ESP-IDF Programming Guide v5.2.1 documentation." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/sleep_modes.html

[76] M. Tanveer, G. Abbas, Z. H. Abbas, M. Waqas, F. Muhammad, and S. Kim, "S6AE: Securing 6LoWPAN Using Authenticated Encryption Scheme," *Sensors*, vol. 20, no. 9, p. 2707, Jan. 2020, number: 9 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1424-8220/20/9/2707

[77] T. M. Jois, G. Beck, S. Belikovetsky, J. Carrigan, A. Chator, L. Kostick, M. Zinkus, G. Kaptchuk, and A. D. Rubin, "SocIoTy: Practical Cryptography in Smart Home Contexts," *Proceedings on Privacy Enhancing Technologies*, vol. 2024, no. 1, pp. 447–464, Jan. 2024. [Online]. Available: https://petsymposium.org/popets/2024/popets-2024-0026.php

[78] S. Sistu, Q. Liu, T. Ozcelebi, E. Dijk, and T. Zotti, "Performance Evaluation of Thread Protocol based Wireless Mesh Networks for Lighting Systems," in *2019 International Symposium on Networks, Computers and Communications (ISNCC)*. Istanbul, Turkey: IEEE, Jun. 2019, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8909109/

[79] D. Lan, Z. Pang, C. Fischione, Y. Liu, A. Taherkordi, and F. Eliassen, "Latency Analysis of Wireless Networks for Proximity Services in Smart Home and Building Automation: The Case of Thread," *IEEE Access*, vol. 7, pp. 4856–4867, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8586865/

[80] H.-S. Kim, S. Kumar, and D. E. Culler, "Thread/OpenThread: A Compromise in Low-Power Wireless Multihop Network Architecture for the Internet of Things," *IEEE Communications Magazine*, vol. 57, no. 7, pp. 55–61, Jul. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8767079/

[81] Silicon Labs, "AN1141: Thread Mesh Network Performance." [Online]. Available: https://www.silabs.com/documents/login/application-notes/an1141-thread-mesh-network-performance.pdf

[82] ——, "AN1142: Mesh Network Performance Comparison." [Online]. Available: https://www.silabs.com/documents/public/application-notes/an1142-mesh-network-performance-comparison.pdf

[83] Eva Azoidou, "Battery Lifetime Modelling and Validation of Wireless Building Automation Devices in Thread," Master's thesis, KTH Royal Institute of Technology School of Electrical Engineering and Computer Science, Stockholm, Sweden, 2016. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1067124/FULLTEXT01.pdf

[84] E. Azoidou, Z. Pang, Y. Liu, D. Lan, G. Bag, and S. Gong, "Battery Lifetime Modeling and Validation of Wireless Building Automation Devices in Thread," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 2869–2880, Jul. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8106811/

[85] "espressif/openthread: Espressif fork of OpenThread project, used to maintain ESP-specific patches and release branches." [Online]. Available: https://github.com/espressif/openthread

[86] "UCSC Thread-ASCON Implementing ASCON in OpenThread," 2024. [Online]. Available: https://github.com/UCSC-ThreadAscon

[87] Nordic Semiconductors, "nRF52840 Dongle Product Brief." [Online]. Available: https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52840-Dongle-product-brief.pdf

[88] ——, "nRF52840 DK: Bluetooth 5.3 SoC supporting Bluetooth Low Energy, Bluetooth mesh, NFC, Thread and Zigbee." [Online]. Available: https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52840-DK-product-brief.pdf

[89] "nRF Sniffer for 802.15.4." [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nRF-Sniffer-for-802154

[90] Nordic Semiconductors, "Installing the nRF Sniffer capture plugin in Wireshark," Apr. 2024. [Online]. Available: https://docs.nordicsemi.com/bundle/ug_sniffer_802154/page/UG/sniffer_802154/installing_sniffer_802154_plugin.html

[91] ——, "Configuring decryption keys for Thread," Apr. 2024. [Online]. Available: https://docs.nordicsemi.com/bundle/ug_sniffer_802154/page/UG/sniffer_802154/configuring_sniffer_802154_thread_keys.html

[92] L. Casillas, "Answer to "What is the purpose of associated authenticated data in AEAD?"," Feb. 2018. [Online]. Available: https://security.stackexchange.com/a/179279

[93] poncho, "Answer to "What is Associated Data in AEAD?"," Sep. 2020. [Online]. Available: https://crypto.stackexchange.com/a/84054

[94] "UCSC-ThreadAscon/openthread: Fork of the Espressif fork of OpenThread project. Implements the ASCON cipher in OpenThread for ESP32 devices." [Online]. Available: https://github.com/UCSC-ThreadAscon/openthread/tree/main

[95] "UCSC-ThreadAscon/esp-idf: UCSC Thread ASCON fork of Espressif IoT Development Framework. Official development framework for Espressif SoCs." [Online]. Available: https://github.com/UCSC-ThreadAscon/esp-idf

[96] Synchronized sleepy end devices (SSEDs) | configuring sleepy devices | OpenThread | latest | silicon labs. [Online]. Available: https://docs.silabs.com/openthread/latest/openthread-sleepy-devices/03-synchronized-sleepy-end-devices-sseds

[97] "UCSC-ThreadAscon/energy-usage-sed-simple," Apr. 2025, original-date: 2024-07-06T05:48:53Z. [Online]. Available: https://github.com/UCSC-ThreadAscon/energy-usage-sed-simple

[98] "UCSC-ThreadAscon/br-energy," Apr. 2025, original-date: 2024-04-02T19:04:05Z. [Online]. Available: https://github.com/UCSC-ThreadAscon/br-energy

[99] "UCSC-ThreadAscon/br-network-performance," Apr. 2025, original-date: 2024-05-15T19:10:53Z. [Online]. Available: https://github.com/UCSC-ThreadAscon/br-network-performance

[100] "UCSC-ThreadAscon/network-performance-ftd," Mar. 2025, original-date: 2024-03-17T02:35:42Z. [Online]. Available: https://github.com/UCSC-ThreadAscon/network-performance-ftd

[101] "UCSC-ThreadAscon/correctness-tests," May 2024, original-date: 2024-05-01T03:00:32Z. [Online]. Available: https://github.com/UCSC-ThreadAscon/correctness-tests

[102] Maria Eichlseder, Vladimir Vissoultchev, and Armando Faz, "meichlseder/pyascon," May 2025, original-date: 2014-04-09T14:16:25Z. [Online]. Available: https://github.com/meichlseder/pyascon

[103] Eve Systems, "Eve Energy | evehome.com." [Online]. Available: https://www.evehome.com/en-us/eve-energy

[104] ——, "Discover Thread | evehome.com." [Online]. Available: https://www.evehome.com/en-us/thread

[105] Belkin, "Wemo Smart Plug with Thread | Belkin." [Online]. Available: https://www.belkin.com/smart-plug-with-thread/WSP100.html

[106] "Wemo Smart Plug with Thread - Smart Outlet for Apple HomeKit - Smart Home Products, Smart Home Lighting, Smart Home Gadgets - Homekit Smart Plug - Tech Gifts - Works W/ Apple iPhone, Easy NFC Set Up : Tools & Home Improvement." [Online]. Available: https://www.amazon.com/Wemo-Smart-Plug-Thread-Products/dp/B09T4S3QKC

[107] J. Hui, "Network Time Sync, what does the offset represent and how to apply across nodes · Issue #5146 · openthread/openthread," Jun. 2020. [Online]. Available: https://github.com/openthread/openthread/issues/5146#issuecomment-878626261

[108] ——, "Network time synchronization · openthread · Discussion #8752," Feb. 2023. [Online]. Available: https://github.com/orgs/openthread/discussions/8752#discussioncomment-4962266

[109] ——, "Network time synchronization · openthread · Discussion #8752," Feb. 2023. [Online]. Available: https://github.com/orgs/openthread/discussions/8752#discussioncomment-4988923

[110] OpenThread, "Network Time Synchronization." [Online]. Available: https://openthread.io/reference/group/api-network-time

[111] M. Wei, "OpenThread Network Time Synchronization Fails to Build on the ESP Thread Border Router (IDFGH-13108) · Issue #14055 · espressif/esp-idf," Jun. 2024. [Online]. Available: https://github.com/espressif/esp-idf/issues/14055#issuecomment-2195636821

[112] Google Nest Help, "How Google products use Thread." [Online]. Available: https://support.google.com/googlenest/answer/9249088#zippy=%2Cverify-your-devices-thread-connection

[113] "What is the Average Room Temperature?" [Online]. Available: https://www.adt.com/resources/average-room-temperature

[114] Eve Systems, "Eve Motion." [Online]. Available: https://www.evehome.com/en-us/eve-motion

[115] ——, "Eve Room | evehome.com." [Online]. Available: https://www.evehome.com/en-us/eve-room

[116] Aqara, "Door and Window Sensor P2." [Online]. Available: https://www.aqara.com/us/product/door-and-window-sensor-p2/

[117] Eve Systems, "Eve Door & Window." [Online]. Available: https://www.evehome.com/en/eve-door-window

[118] A. Betzler, C. Gomez, I. Demirkol, and J. Paradells, "A Holistic Approach to ZigBee Performance Enhancement for Home Automation Networks," *Sensors*, vol. 14, no. 8, pp. 14 932–14 970, Aug. 2014, number: 8 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1424-8220/14/8/14932

[119] Espressif, "Amazon.com: ESP Thread Border Router/Zigbee Gateway Board : Electronics." [Online]. Available: https://www.amazon.com/Thread-Border-Router-Zigbee-Gateway/dp/B0C89H9MJ8

[120] ——, "Amazon.com: ESP32-H2-DevKitM-1-N4 Development Board : Electronics." [Online]. Available: https://www.amazon.com/Espressif-ESP32-H2-DevKitM-1-N4-Development-Board/dp/B0BWM83LMF

[121] Dell, "Dell OptiPlex 9020." [Online]. Available: https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/optiplex-9020-micro-technical-spec-sheet.pdf

[122] STG USA, "Amazon.com: Dell Optiplex 9020 Small Form Factor Desktop with Intel Core i7-4770 Upto 3.9GHz, HD Graphics 4600 4K Support, 32GB RAM, 1TB SSD, DisplayPort, HDMI, Wi-Fi, Bluetooth - Windows 10 Pro (Renewed) : Electronics." [Online]. Available: https://www.amazon.com/dp/B08BJDFZRF?ref_=ppx_hzsearch_conn_dt_b_fed_asin_title_3&th=1

[123] Vangree, "Amazon.com: Powered USB Hub, VANGREE 17-Port 90W USB 3.0 Hub (10 USB 3.0 Ports+3 QC24W Fast Charging Ports+2 USB-C 3.0 Ports+SD/TF Card Reader), Individual On/Off Switches, 12V/7.5A Power Adapter for Laptop, PC : Electronics." [Online]. Available: https://www.amazon.com/dp/B0BS1TMYY6?ref_=ppx_hzsearch_conn_dt_b_fed_asin_title_2&th=1

[124] Wenter, "Amazon.com: Powered USB 3.0 Hub, Wenter 5-Port USB Hub Splitter (4 Faster Data Transfer Ports+ 1 Type-C Charging Ports) with Individual LED On/Off Switches, USB Hub 3.0 Powered with Power Adapter for Mac, PC : Electronics." [Online]. Available: https://www.amazon.com/dp/B0C1H46YM7?ref_=ppx_hzsearch_conn_dt_b_fed_asin_title_1&th=1

[125] TP-Link, "Kasa Smart WiFi Plug Mini." [Online]. Available: https://www.tp-link.com/us/home-networking/smart-plug/ep10/

[126] Espressif, "Current Consumption Measurement of Modules - ESP32 - — ESP-IDF Programming Guide latest documentation." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/current-consumption-measurement-modules.html

[127] Nordic Semiconductors, "Power Profiler Kit II," Aug. 2022.

[128] "ESP-IDF ot_sleepy_device." [Online]. Available: https://github.com/espressif/esp-idf/tree/master/examples/openthread/ot_sleepy_device/deep_sleep

[129] "What does the J5 Pin Specifically Do on the ESP32-H2?" May 2024. [Online]. Available: https://www.esp32.com/viewtopic.php?f=12&t=40052

[130] Lenovo, "Product Overview - Yoga 730-15IWL - Lenovo Support US," Sep. 2018. [Online]. Available: https://support.lenovo.com/us/en/solutions/pd500250-product-overview-yoga-730-15iwl

[131] Espressif, "Universal Asynchronous Receiver/Transmitter (UART) - ESP32 - — ESP-IDF Programming Guide latest documentation." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/uart.html

[132] Kamil, "Answer to "Working out mAh from current and time"," Apr. 2014. [Online]. Available: https://electronics.stackexchange.com/a/107078

[133] "Amazon.com : AA Battery mAh." [Online]. Available: https://www.amazon.com/s?k=AA+Battery+mAh&ref=nb_sb_noss

[134] Amazon, "Amazon.com: Amazon Basics Rechargeable AA Batteries, 8-Pack, 2400 mAh, NiMH High-Capacity Batteries, Recharge up to 400x Times, Pre-Charged : Health & Household." [Online]. Available: https://www.amazon.com/AmazonBasics-High-Capacity-Rechargeable-Batteries-Pre-charged/dp/B00HZV9WTM

[135] OpenThread, "Radio Configuration," Feb. 2025. [Online]. Available: https://openthread.io/reference/group/radio-config

[136] Simeon Tran, "How to Modify the 802.15.4 TX Power on an ESP32-H2? - ESP32 Forum," Mar. 2024. [Online]. Available: https://esp32.com/viewtopic.php?t=38671

[137] Espressif, "ESP Wireless Transmission Power Configuration - - — ESP-Techpedia latest documentation." [Online]. Available: https://docs.espressif.com/projects/esp-techpedia/en/latest/esp-friends/advanced-development/performance/modify-tx-power.html

[138] Zigbee Alliance, "ZigBee Specification," Apr. 2017. [Online]. Available: https://csa-iot.org/developer-resource/specifications-download-request/

[139] OpenThread, "otCoapTxParameters Struct Reference | OpenThread," Jan. 2024. [Online]. Available: https://openthread.io/reference/struct/ot-coap-tx-parameters

[140] ——, "CoAP | OpenThread." [Online]. Available: https://openthread.io/reference/group/api-coap

[141] Sprite, "Restart ESP Programmatically - ESP32 Forum," Feb. 2018. [Online]. Available: https://esp32.com/viewtopic.php?t=4706#p20412

[142] horace, "How to distinguish reset by power on from reset by software command ? - Projects / General Guidance," Aug. 2023, section: Projects. [Online]. Available: https://forum.arduino.cc/t/how-to-distinguish-reset-by-power-on-from-reset-by-software-command/1160400/9

[143] Espressif, "Miscellaneous System APIs - ESP32 - — ESP-IDF Programming Guide latest documentation." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/misc_system_api.html#_CPPv411esp_restartv