



TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF
ELECTRONIC & COMPUTER ENGINEERING
ELECTRONICS & COMPUTER
ARCHITECTURE DIVISION

**A lightweight and secure MQTT implementation
for Wireless Sensor Nodes**

by

Sotirios Katsikeas

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DIPLOMA OF ELECTRONIC
AND COMPUTER ENGINEERING

Chania, June 2016

THESIS COMMITTEE

Associate Professor Ioannis Papaefstathiou, Thesis Supervisor

Professor Apostolos Dollas

Associate Professor Antonios Deligiannakis

Acknowledgments

With my academic life at the School of Electronic and Computer Engineering in Technical University of Crete coming to an end, I would like to express my gratitude to a lot of people for their support throughout my studies.

Firstly, I would like to thank my thesis supervisor, Associate Professor Ioannis Papaefstathiou, for his trust, guidance, support and cooperation from the beginning till the end of my thesis and also for offering me the opportunity to broaden my knowledge on the subject of Information Security.

Additionally, I would also like to specially thank Dr. Konstantinos Fysarakis, for all his help, guidance and for his patience to support me every time I “bothered” him.

Last but not least, I would like to thank my family for their continuous support the entire time of my studies, my life companion Emily for her unlimited love, emotional support and tolerance during the hard times of my academic life and of course my friends who are next to me all those years.

Abstract

During the past years, with the adoption of the IPv6 protocol which has provided us with a vast amount of addresses available for use at anything we might want, together with the need to have every single of our devices and appliances connected to the Internet in order to be controlled by our mobile devices, the technological term “Internet of Things”, also abbreviated as IoT, has resurfaced. That is an indeed very trending topic today and it is estimated that by 2020, almost 50 billion “things” will be connected to the Internet. However, having all these devices connected to the Internet, which, as we already know, is not exactly what one would deem a “safe place”, harbors a great deal of dangers and vulnerabilities that could render any device accessible to a malicious user. In order to protect our “things” and our transferred data, we need to implement security mechanisms. The problem is that due to the constrained hardware and resources of these devices, not every existing security mechanism can be successfully implemented on them. In this thesis, the MQTT protocol was selected, among other IoT communication protocols, as a foundation, on top of which we will develop different lightweight security implementations for Wireless Sensors Nodes running the Contiki OS for Internet of Things. However, given the importance of security in IoT and because of the many options in relation to where and how to implement security mechanisms in MQTT, a comparison and an assessment of the performance of these options will be made in order to designate the best one.

Keywords: Internet of Things, IoT, MQTT, information security, encryption, authentication, secure communication

Table of Contents

Acknowledgments.....	2
Abstract	3
Table of Contents	4
List of Figures.....	6
List of Tables	7
1. Introduction	8
1.1. Purpose.....	9
1.2. Limitations	9
1.3. Method	10
2. Technical Background.....	11
2.1. Wireless Sensor Networks (WSNs)	11
2.1.1. The WSNs and other IoT communication models.....	12
2.2. IoT Communications	13
2.2.1. IEEE 802.15.4.....	13
2.2.2. Internet Protocol version 6 (IPv6).....	14
2.2.3. 6LoWPAN.....	14
2.2.4. TCP.....	14
2.3. IoT Security.....	15
2.3.1. Security on the IEEE 802.15.4 (Data Link layer).....	15
2.3.2. Security on the Network Layer.....	16
2.3.3. Security on the Transport Layer	16
2.3.4. The Advanced Encryption Standard (AES)	17
2.3.5. Security on the Application Layer.....	18
2.4. IoT Protocols Stack	18
2.4.1. Overview of protocols/comparison.....	19
3. The MQTT protocol	23
3.1. Security on MQTT	25
3.1.1. Authentication on MQTT	25
3.1.2. Authorization on MQTT	25
3.1.3. Alternative to using TLS.....	26

3.1.4.	Other notes on MQTT security	27
4.	Implementation	28
4.1.	Tools Utilized	28
4.1.1.	Contiki OS as an Operating System for IoT.....	28
4.1.2.	6lbr as a 6LowPAN Border Router	29
4.1.3.	Mosquitto as a MQTT broker	31
4.1.4.	Zolertia Z1 as a hardware development platform	32
4.2.	General notes on implementation	33
4.2.1.	Testing setup topologies	34
4.2.2.	Configuration of the WSN notes	35
4.2.3.	6lbr configuration	35
4.2.4.	MQTT client source code	36
4.3.	Secure and lightweight MQTT implementations	37
4.3.1.	Option 1 – Payload encryption with AES	37
4.3.2.	Option 2 – Payload encryption with AES-CBC	37
4.3.3.	Option 3 – Payload authenticated encryption with AES-OCB... ..	38
4.3.4.	Option 4 – Link layer encryption with CCM*	38
4.3.5.	Implementation diagrams	39
5.	Performance Assessment/Evaluation	40
5.1.	Power specifications of Zolertia Z1	42
5.2.	Evaluation of option 0 (simple MQTT client)	43
5.3.	Evaluation of option 1 (single block AES).....	45
5.4.	Evaluation of option 2 (AES-CBC)	47
5.5.	Evaluation of option 3 (AES-OCB)	49
5.6.	Evaluation of option 4 (Link Layer Security)	52
5.7.	Comparison.....	54
6.	Conclusions and Open Issues.....	60
6.1.	Conclusions.....	60
6.2.	Problems encountered.....	60
6.3.	Open Issues.....	62
	Bibliography.....	63
	Annex A.....	68

List of Figures

Figure 2.1: Example of WSN setup 12

Figure 2.2: The IoT stack..... 18

Figure 3.1: A simple MQTT publish/subscribe example 23

Figure 3.2: The End-to-End encryption scenario [31] 26

Figure 3.3: The Client-to-Broker encryption scenario [31]..... 27

Figure 4.1: 6lbr border router diagram [9]..... 30

Figure 4.2: mqtt-spy (MQTT client) showing broker statistics 31

Figure 4.3: A Zolertia Z1 mote..... 32

Figure 4.4: The topology of the virtual environment..... 34

Figure 4.5: The topology of the real world environment..... 34

Figure 4.6: Real motes deployed during evaluation 35

Figure 4.7: 6lbr deployed on Raspberry Pi 36

Figure 4.8: The simple MQTT client running on Cooja 37

Figure 4.9: The MQTT publish action diagram 39

Figure 4.10: The publish to a subscriber client action diagram..... 39

Figure 5.1: The ACK mechanism for latency measurements 41

Figure 5.2: Option 0 – Publishing mote’s consumption graph..... 44

Figure 5.3: Option 0 – Subscribed mote’s consumption graph..... 44

Figure 5.4: Option 1 – Publishing mote’s consumption graph 46

Figure 5.5: Option 1 – Subscribed mote’s consumption graph..... 46

Figure 5.6: Option 2 – Publishing mote’s consumption graph 48

Figure 5.7: Option 2 – Subscribed mote’s consumption graph..... 48

Figure 5.8: Performance evaluation of AES-OCB on Cooja..... 49

Figure 5.9: Option 3 – Publishing mote’s consumption graph 50

Figure 5.10: Option 3 – Subscribed mote’s consumption graph..... 51

Figure 5.11: Option 4 – Publishing mote’s consumption graph 53

Figure 5.12: Option 4 – Subscribed mote’s consumption graph..... 53

Figure 5.13: Publishing mote’s consumption comparison graph 54

Figure 5.14: Subscribed mote’s consumption comparison graph..... 55

Figure 5.15: Average message latency comparison graph 55

Figure 5.16: Publishing mote’s program size comparison graph 57

Figure 5.17: Receiving mote’s program size comparison graph..... 58

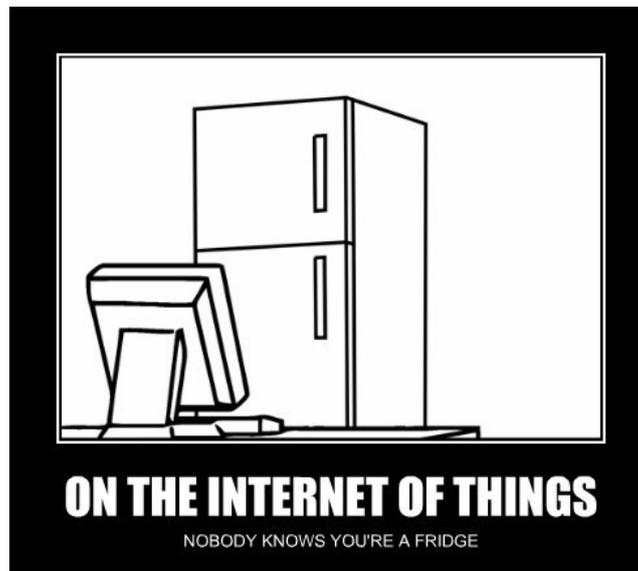
Figure 5.18: Average message latency vs payload size comparison graph.... 59

List of Tables

Table 2.1: The different security modes available on IEEE 802.15.4	16
Table 2.2: Feature comparison of the main IoT protocols	21
Table 4.1: OS requirements support comparison [33].....	28
Table 4.2: The versions of the tools used in implementation phase	33
Table 5.1: Power specifications of Zolertia Z1.....	42
Table 5.2: Option 0 - Average power consumptions	43
Table 5.3: Option 0 - Average radio duty cycle.....	43
Table 5.4: Option 0 – Program size	43
Table 5.5: Option 1 - Average power consumptions	45
Table 5.6: Option 1 - Average radio duty cycle.....	45
Table 5.7: Option 1 – Program size	45
Table 5.8: Option 2 - Average power consumptions.....	47
Table 5.9: Option 2 - Average radio duty cycle.....	47
Table 5.10: Option 2 – Program size	47
Table 5.11: Option 3 - Average power consumptions.....	49
Table 5.12: Option 3 - Average radio duty cycle	49
Table 5.13: Option 3 – Program size	50
Table 5.14: Option 4 - Average power consumptions	52
Table 5.15: Option 4 - Average radio duty cycle.....	52
Table 5.13: Option 4 – Program size	52

1. Introduction

In today's highly advanced technological world, every day more and more devices and appliances are being replaced by their new generation counterparts that, while, in effect, have the same basic use, they are now "smart". The term "smart", itself, denotes that these devices can now connect to the user's other devices and exchange information in order to adjust their functionality according to the user's needs or even enable the user to control and monitor all of them from a single point. All of the above concur to one thing, that all these "smart" devices/appliances need to be connected to a large network and be accessible from everywhere. The best network candidate for the above description is no other than the Internet. At this point one can easily imagine how the term Internet of Things comes about. There are obviously a lot of smart things that are connected to the Internet!



A well-known illustration of the Internet of Things

In a more formal way, Internet of Things (IoT) is a technological concept aiming at connecting all things to the Internet [1]. Another approach from the Cisco Internet Business Solutions Group (IBSG), describes the IoT as simply the point in time when more "things or objects" were connected to the Internet than people [2]. It must be stated, however, that there is no restriction on the type and complexity of the aforementioned "things"; ranging from simple devices, such as a coffee maker, for example, to more complex machines like cars or airplanes. There is also no limit on the amount of them. A recent research by Cisco on 2011 estimated that the IoT will consist of almost 50 billion objects by the year 2020 [3].

One could easily say that IoT will change our lives the same way as smartphones did back in 2007. In general, nowadays, smartphones and mobile

devices allow us constant and direct access to the Internet and to a plethora of information from anywhere in the world. IoT, it could be argued, will do the same for devices, enabling them to be constantly connected to the Internet. That could be really useful for our everyday lives as it could be the foundation of autonomous systems that will save us time and energy and could also help us solve today's technological problems using its distributed nature.

Unfortunately, there is an important, yet at the same time, simple problem that comes together with the concept of the Internet of Things: security! Due to the fact that all these devices will be connected to the internet, there must be security mechanisms that will not allow malicious access to them, and simultaneously protect the exchanged data. That is an already solved problem for modern PCs and mobile devices, we cannot, however, say the same for the devices that comprise the IoT. Such devices, in their majority, have a very constrained set of resources (small amounts of RAM, ROM and power) and capabilities (low processing power) that could not withstand security implementations used on other more powerful devices such as smartphones and tablets. For example, the widely used IPsec protocol suite could not be directly used on many IoT devices because of the above constraints and must be modified in order to be more lightweight [4, 5]. As a result, new, redesigned or a mixture of previous security mechanisms must be used for these devices.

1.1. Purpose

The purpose of this thesis is to implement a lightweight and secure implementation for the MQTT [6] communication protocol on Wireless Sensors Nodes that are running the Contiki OS [7] for Internet of Things. This shall be done by evaluating the already available security mechanisms, and, then, by developing different secure MQTT implementations based on the most promising of them. Finally, the different implementations will be evaluated using performance measurement data, and a decision on which one of the implementations is the best option for such constrained devices will be made. In more detail, the physical, link, and application layers of the Open Systems Interconnection model (OSI model) will be examined, in order to find suitable technologies on the market.

1.2. Limitations

This thesis is focused on the MQTT communication protocol and on devices running the Contiki OS and will not cover in depth other communication protocols, such as CoAP and DPWS or other operating systems such as Tiny OS. In this thesis, a comparison between the features of the major communication protocols for the IoT will be drawn and the choice of MQTT will be explained. Furthermore, a final note on the used hardware must be

mentioned. For the implementation and testing part of this thesis, only Zolertia Z1 [8] was used as hardware development platform, because of its availability at that current time. It must, however, be highlighted that the secure MQTT implementations developed for this thesis are compatible with other platforms running the Contiki OS, and only some minor configuration changes should be needed on some cases. Finally, for real life testing of the developed secure MQTT mechanisms 6lbr [9] was used as a 6LoWPAN Border Router for connecting the 802.15.4 network with the IPv6 LAN.

1.3. Method

To make sure that the right security mechanisms were selected and investigated, the first phase of this thesis was a literature study. The study served as a foundation when developing and performing the evaluation of the security methods. After the literature study, a selection process was performed, where the most promising mechanisms were examined in further detail and brought into the development phase. At this phase, a research for the best MQTT client implementation for Contiki OS was also being carried out. This process included the selection of mechanisms that were already implemented for use on such constrained devices. In the development phase, the chosen platform was programmed with the developed code that included the MQTT client, as well as the security mechanism and was prepared for testing; it was then tested firstly for stability and then according to energy consumption, latency and program size on both ROM and RAM of the development platform.

2. Technical Background

Before we even begin to dive into the technical background needed for the Internet of Things, let us first discuss the origins of that term. The term itself has existed for a long time, despite having only resurfaced during the past years.

The term “Internet of Things” (IoT) was first used in 1999 by British technology pioneer Kevin Ashton [10], to describe a system in which objects in the physical world could be connected to the Internet by sensors. More specifically, he was occupied with Radio-Frequency Identification (RFID) [11] tags and its use in corporate supply chains connected to the Internet for tracking and counting goods without the need of human intervention. Today, the Internet of Things has become a popular term for describing scenarios in which Internet connectivity and computing capability extend to a variety of objects, devices, sensors, and everyday items [12].

But why is the Internet of Things so popular today? That is a direct result of a recent advance in several fields of technology that have made possible to interconnect more and smaller devices, cheaply and easily. One characteristic example of the aforementioned advance in technology, is the widespread adoption of IP based networking and the use of the newly designed IPv6 [13] protocol that provides us with a vast amount of available IP addresses. Another significant advance, that is also described by Moore’s Law [14], is the ability to produce more powerful and smaller computer hardware electronics that have lower power consumption approximately every two years. All this, coupled with the advances in Data Analytics, -with better algorithms and the use of distributed systems-, and the rise of the Cloud Computing, have made the Internet of Things a really popular topic that has attracted a great amount of research. As some last words about the IoT, it must be emphasized that its use is not restricted to simple scenarios of use, as is, for example, home automation, or in “smart” devices, but is also used in large scale scenarios in industries, also known as “Industrial IoT”. For that reason, a great lot of work is currently underway on standardizing the protocols that are used in IoT by the IEEE SA [15], the AllSeen [16], the OASIS [17], the IIC [18] and some other organizations. The problem of the aforementioned standardizing for the IoT is that every organization proposes different standards which, while similar in many ways, are not identical, and therefore, incompatible [19].

2.1. Wireless Sensor Networks (WSNs)

Today sensors are everywhere; in our cars, in our mobile devices and even in our homes. Even though sensors surround us for many years, the concept of having wireless networks of sensors is relatively new. While research on

Wireless Sensor Networks (WSN) started back in the 1980s, a significant increase in its use has only been noticed from 2001 and onwards [20] due to the already mentioned advances in technology. Wireless Sensor Networks are used today as a common foundation for constructing IoT applications and systems mainly because of its small form factor and low power needs.

A WSN can in general be described as a network of nodes (or otherwise a group of sensors) that are scattered in a certain area and, cooperatively sense the environment and send the information to a central node that processes it and may produce some control commands and/or enable interaction between persons or computers and the surrounding environment.

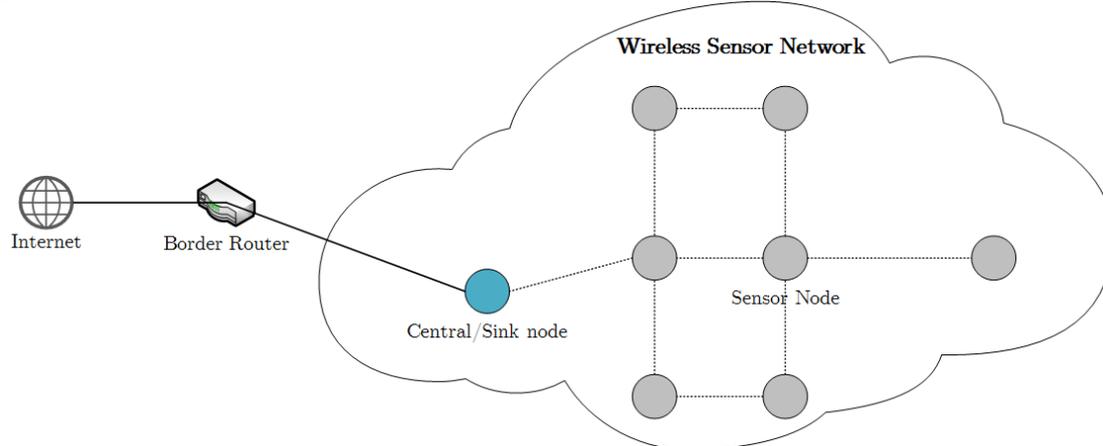


Figure 2.1: Example of WSN setup

WSN nodes (or also called notes) can be placed in the area according to many different topologies. As one can observe from the above figure, in this specific WSN topology it is not mandatory that every node has a network link with the sink node, as it could be reached through other nodes with multiple hops. This example shows a multi-hop wireless mesh network topology. This kind offers the greater scalability among other topologies. Every node maintains a link to other nodes within range (i.e. its neighbors) and the decision on which one to use is reached by the used routing protocol according to the number of hops and the latency of every route. The nodes of a WSN cannot always be of the same type, but they must support the same standards and protocols in order to be able to communicate with each other.

The border router is the connection link between the Internet and the WSN mesh network. It converts IPv6 frames into mesh networks frames because of the differences they have in order for the latter to be as small as possible for use in constrained devices such as WSN nodes.

2.1.1. The WSNs and other IoT communication models

Wireless Sensor Networks (WSN) are following the Device-to-Gateway communication model where the devices are the WSN nodes and the gateway is the combination of the Sink node and the Border Router. That model is used

in WSNs because of its feature ability to overcome proprietary device restrictions in connecting IoT devices. This, of course, means that device interoperability and standards are very important considerations in the design and development of interconnected IoT systems. According to the *Internet Society*, there are three more communications models for the IoT that will only be mentioned epigrammatically, since the focus of this thesis is only on WSNs used in IoT. The first one is the Device-to-Device (D2D) communication model also referred as Machine to Machine (M2M) in industrial use and describes two or more devices that directly connect and communicate between one another, rather than through an intermediate application server or base station. Another one is the Device-to-Cloud communication model where the IoT device connects directly to an Internet cloud service like an application service provider to exchange data and control message traffic. Finally, there is the Back-End Data-Sharing model which is a more complex model that enables multiple applications to process the shared data coming from IoT devices and produce a collective result.

2.2. IoT Communications

With only just a quick research a variety of communication standards and protocols used on the Internet of Things can be found; below follows the description of the most popular of these, and are the ones that will be used in this thesis.

2.2.1. IEEE 802.15.4

One of the most widely used mesh network communication standards is IEEE 802.15.4 [21]. It defines the Physical (Layer 1) layer and the Data-Link (Layer 2) layer, and more specifically the Medium Access Control (MAC) sublayer of that layer, of the OSI model. The first revision of the 802.15.4 standard was released in May 2003. Today, several standardized and proprietary network (or mesh) layer protocols run over 802.15.4 based networks, including IEEE 802.15.5, ZigBee, 6LoWPAN, WirelessHART, and ISA100.11a.

This standard is designed with one simple target in mind; to achieve the lowest power consumption. According to this, it is designed for use in low cost, low speed communication between low power devices. As a result, this standard describes a slower and smaller range wireless communication method compared to the IEEE 802.11, which is the Wi-Fi we all use every day, with the tradeoff of significant lower power consumption. The basic specifications include a 10 meter (maximum 20 meters) communications range with a transfer rate of up to 250 kbps with other options of 100kbps, 40 kbps, and 20 kbps available as well. Devices are able to communicate in one of the three possible frequency bands for operation (868/915/2450 MHz). Finally, it uses CSMA/CA protocol

[22] as a MAC mechanism and two addressing modes; 16-bit short and 64-bit IEEE addressing.

2.2.2. Internet Protocol version 6 (IPv6)

Internet Protocol version 6 (IPv6) [13] is the most recent version of the Internet Protocol (IP); the communications protocol that provides an identification and location system for computers on networks, and routes traffic across the Internet. IPv6 was developed by the *Internet Engineering Task Force* (IETF) to deal with the problem of IPv4 address exhaustion. IPv6 uses 128 bit addresses, theoretically allowing 2^{128} , or approximately 340 trillion addresses. As a result, it is safe to conclude, that IPv6 is, as one may put it, over satisfying the need of addresses not only for our computing devices but also for all the IoT devices that will ever be available. Finally, IPv6 introduces a new set of features that were not implemented on the previous version (IPv4) that make the IP protocol more reliable and easy to configure. Two examples are the simplified address assignment with the use of a mechanism called Stateless Address Autoconfiguration (SLAAC) and the transfer of the responsibility for packet fragmentation from routers to the end points.

2.2.3. 6LoWPAN

6LoWPAN is an abbreviation of "IPv6 over Low Power Wireless Personal Area Networks" [23]. The 6LoWPAN concept comes from the idea that "the Internet Protocol could and should be applied even to the smallest devices" [24] and that low-power devices with limited processing capabilities should be able to participate in the Internet of Things [25]. 6LoWPAN enables constrained devices that are unable to handle the traditional IP stack to function, and connect to a IPv6 network, such as the Internet. The mapping from the IPv6 to the IEEE 802.15.4 was not an easy task to carry out, because of the many differences mainly in terms of packet size, address resolution and maximum transmission unit (MTU) of the two networks. It was finally achieved by compressing and encapsulating the IPv6 and UDP headers.

2.2.4. TCP

Due to the fact that on the Transport Layer the common TCP protocol is used, the author of this thesis deemed it redundant to delve into any further details about it. Nevertheless, for more information, the reader is advised to consult Blank (2006) [26].

2.3. IoT Security

Now that all the relevant technologies that will be used have been presented and their main features have been described, it is time to overview the security options and mechanisms that are available for every one of them, starting from the lower levels of the OSI and progressing upwards. Before that, and as an important note, it must be emphasized that the establishment of security on different layers of the OSI Model inherits different, and sometimes unique, security features.

2.3.1. Security on the IEEE 802.15.4 (Data Link layer)

In order to be a complete standard, IEEE 802.15.4 was equipped with some strong security mechanisms that are also able to run on constrained devices. The main security mechanisms that are present in the IEEE 802.15.4 standard specification are access control; confidentiality; frame integrity (or otherwise authenticity); and protection against replay attacks (also referred to as sequential freshness).

The IEEE 802.15.4 security is handled at the MAC sublayer of the Data Link layer. The security mechanisms to be used are specified at the application layer by setting some control parameters. The IEEE 802.15.4 specification has a choice of security modes that control different security levels [28]. Each security mode has different security properties, protection levels, and frame formats. The following table (table 2.1) shows the available security suites and the value of the corresponding parameter (here, the Security Level field parameter) that corresponds to either one of them. The 0x00 value sets no encryption, so that neither the data is encrypted (no data confidentiality), nor the data's authenticity is validated. From the 0x01 to 0x03, the data is authenticated using the encrypted Message Authentication Code (MAC), but not encrypted. The value 0x04 encrypts the payload ensuring only data confidentiality. And finally, the 0x05 to 0x07 range ensures both data confidentiality and authenticity. The encryption modes supported by IEEE 802.15.4 are based on the AES (Advanced Encryption Standard) encryption suite and more specifically on the AES-CBC and AES-CCM* (pronounced CCM-star) block cipher modes of operation, that will be explained on the last section of this chapter (i.e. section 2.3.4).

Furthermore, establishing security at this layer comes with two advantages: i) the whole encryption/decryption and authentication (if exists) process is much faster than it is in other, higher layers, and ii) it has relatively easier setup, mainly because it comes pre-developed or, often, as an easy to enable feature of the data link protocol. As a conclusion, a good security option is to make use of the security suites provided on IEEE 802.15.4 in order to establish security in our case.

Value	Mode name	Description	Access Control	Confidentiality	Frame Integrity	Seq. Freshness
0x00	Null	No security	No	No	No	No
0x01	AES-CBC-MAC-32	32-bit MAC	Yes	No	Yes	No
0x02	AES-CBC-MAC-64	64-bit MAC	Yes	No	Yes	No
0x03	AES-CBC-MAC-128	128-bit MAC	Yes	No	Yes	No
0x04	AES-CTR	Encryption only	Yes	Yes	No	Yes
0x05	AES-CCM-32	Encryption & 32-bit MAC	Yes	Yes	Yes	Yes
0x06	AES-CCM-64	Encryption & 64-bit MAC	Yes	Yes	Yes	Yes
0x07	AES-CCM-128	Encryption & 128-bit MAC	Yes	Yes	Yes	Yes

Table 2.1: The different security modes available on IEEE 802.15.4

2.3.2. Security on the Network Layer

As with the case of IPv6, here, also on 6LoWPAN, security can be established with the use of IPSec protocol, with the difference that, in order to work together with 6LoWPAN, modifications, such as compression, must be also done on the IPSec header [4, 5]. Because this concept is not currently under active development, it does present some compatibility issues with the version of Contiki OS and the hardware used on this thesis; (the program for IPSec does not fit in ROM of the Zolertia Z1, except when using an under development version of the msp430-gcc compiler that has other bugs). Moreover, due to the limitations of this study’s timeframe as well as the questionable suitability of IPSec for use in IoT applications [55], we will not focus on using that security option for our secure MQTT implementation, but instead will make use of other security mechanisms on the greater layers of the OSI model.

2.3.3. Security on the Transport Layer

The best way to achieve secure communications on this precise layer is by using TLS, if using TCP, or, by using DTLS, if using UDP. The choice of MQTT (which uses TCP) as our Application Layer protocol automatically deprives us of the option of using DTLS. DTLS, however, can be used over TCP, but it requires advanced techniques such as tunneling [29] that are not implementable on such constrained devices. That means that only TLS is left to be used in our case. There is, nevertheless, yet another problem; namely, that, neither TLS could be implemented on our setup, again, due to the constrained nature of such devices. That leaves us with no other choice than to skip this layer altogether and attempt to establish security on the upper application layer.

2.3.4. The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES), also known as Rijndael (its original name), is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES is based on a symmetric key block cipher encryption algorithm, and was designed according to a design principle known as a substitution-permutation network, where multiple consecutive substitutions and permutations (4 steps in total) are performed for multiple rounds on the plaintext to be encrypted. AES has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits.

Because AES is a fixed block size cipher, in order to be able to encrypt data of not exactly 128 bits (16 bytes) in size it must be used according to a specified block cipher mode of operation. Such modes are, among others, the CBC, CTR and CCM* modes mentioned earlier. Some information about them will be presented below, but a more detailed explanation can be found in [43].

Cipher Block Chaining (CBC) mode is the most commonly used mode of operation as it enables encryption of data with any size (with the use of padding when size is smaller than 128 bits) with the characteristic that each ciphertext block depends on all plaintext blocks processed up to that point. That makes this mode secure and strong against replay attacks. Its main disadvantage is that it cannot be parallelized, due to the fact that it is designed to run sequentially on the blocks to be encrypted, but that is not a problem when used in constrained devices that would not benefit from that either way.

Counter (CTR) mode, overcomes the disadvantage of OCB enabling the encryption to be run in different parallel threads. Counter mode turns a block cipher into a stream cipher. A stream cipher encrypts each plaintext digit one at a time with the corresponding digit of the keystream, in order to give a digit of the ciphertext stream.

Finally, CCM* mode is based on the CCM (Counter with CBC-MAC), defined in RFC 3610 [44], which is an authenticated encryption algorithm designed to provide both authentication and confidentiality, and works in an "authenticate-then-encrypt" manner. CCM* is a variation of CCM designed for use in 802.15.4 that includes all the features of CCM, and additionally offers encryption-only and integrity-only capabilities. CCM mode was developed with the idea to be an alternative to OCB mode (another very similar authenticated encryption algorithm); the latter although being significantly faster (approximately two times faster than CCM and has minimal overhead in comparison with simple encryption modes such as CBC), is patented in the U.S. However, a special exemption has been granted, so that OCB mode can be used in software licensed under the GNU General Public License without cost, as well as for any non-commercial, non-governmental application [45].

2.3.5. Security on the Application Layer

Finally, security could also be established on this particular layer, with this option providing some unique features not encountered on other layers. The first one of them, is the ability to achieve true end-to-end encryption; namely, encryption of the transferred packets with the only ones being able to decrypt them to be either the sender or the recipient. Additionally, end-to-end encryption does not suffer from compatibility issues, since the encryption/decryption is carried out on the application layer and therefore, is platform independent. The second major advantage, is the freedom of security mechanisms options. Either an already existing security mechanism could be used, or if there is need into it, a new and custom made mechanism could be developed.

2.4. IoT Protocols Stack

At this point it should be noted that even though IoT is still under development in terms of standardizing and protocol adoption, it has a large set of protocols already available that are actually similar to the ones available for the Internet. The following table contains the majority of the protocols and standards that were encountered while going through the literature and can be described as the IoT stack.

Application Layer	IoT Application					
	HTTP	XMPP	DPWS	SOAP	CoAP	MQTT
Transport Layer	TLS			DTLS		
	TCP			TCP/UDP		
Network Layer	6LoWPAN			IPSec		
	RPL					
IPv6						
Data Link Layer	IEEE 802.15.4	Bluetooth / Bluetooth LE	RFID / NFC	IEEE 802.11 (Wi-Fi)	GSM / LTE	
Physical Layer						

Figure 2.2: The IoT stack

On the above figure the protocols and standards with yellow gradient background are the ones that this thesis will focus on.

2.4.1. Overview of protocols/comparison

Although this thesis will focus on MQTT as an application layer protocol, for the sake of completeness, a comparison between the different application layer protocols shall be made at this point. However, the comparison, will be articulated only between the protocols that use TCP or UDP as their transport protocol, because the other ones using HTTP as transport protocol are not suitable for lightweight implementations on constrained devices due to the overhead introduced by HTTP. As a result, the HTTP itself, and the SOAP protocol will not be included in the comparison.

The first protocol that will be overviewed is DPWS. The Devices Profile for Web Services (DPWS) (also called as Web Services on Devices by Microsoft). was initially introduced in 2004 by Microsoft and the version 1.1 has been an OASIS standard since 2009 [47]. DPWS is very similar to UPnP (Universal Plug and Play), but is mainly used on large scale enterprise networks rather than home networks in which UPnP is preferred. DPWS is also natively built in many versions of Windows. DPWS relies mainly on UDP but also uses TCP for its transport and is a defined set of minimal implementation constrains to enable secure Web Service messaging, discovery, description, and synchronous and asynchronous communication on resource-constrained devices. Because of its Web Services nature, DPWS describes two types of services: hosting services and hosted services. A hosting service is associated with a single device, while a device may accommodate many hosted services. Other services available from DPWS are: Discovery services, Metadata exchange services and Publish/Subscribe event services. DPWS was used in the EU Research Project SOCRADES [48] that focused on implementing, testing and piloting prototypes of DPWS-enabled devices for use in the industrial automation domain and is currently under research for use in “smart” cities, “smart” homes and other applications.

Extensible Messaging and Presence Protocol (XMPP) is another communications protocol used in IoT and is designed for message-oriented middleware based on XML (Extensible Markup Language). It was developed in 1999 by the Jabber open source community for near-real time data exchange. Due to its extensibility it has a lot of uses in Publish/Subscribe and IoT applications. It is also a standard under the Internet Engineering Task Force (IETF) with its latest specification being RFC 7622 [49]. XMPP uses a client-server architecture where clients do not talk directly to each other. Clients have a unique JID (Jabber IDs) in the form of email (e.g. username@server.com) and can also log-in to the same server from different locations/devices called resources. Each resource has a priority among the others and the messages sent to this specific user will only be delivered to the resource/client with the greater

priority. XMPP uses TCP as its native transport protocol with the ability to maintain long-lived TCP connections between the server and the client.

Constrained Application Protocol (CoAP) is an application layer protocol designed for use in constrained devices in order to allow them to communicate interactively over the Internet. CoAP is designed with simplified integration of WSN nodes into the web, as its target, while also retaining some basic features such as multicast support, very low overhead, and simplicity. A mapping of CoAP to HTTP is also defined enabling access to CoAP resources via HTTP, as simple as loading a website on a browser. It follows the request/response model, where a client interacts with the server using a subset of the HTTP methods that are: GET, PUT, POST and DELETE. CoAP is specified on RFC 7525 [50], while other extensions are currently under the progress of the standardization process. CoAP is probably the most popular application layer protocol used in IoT with numerous examples in domains such as smart homes, mobile IoT deployments, cloud services, healthcare, smart cities, and industrial WSNs.

The MQTT protocol and its features will be extensively described in a following chapter but a comparison between the aforementioned protocols together with MQTT can be found on the below table.

	DPWS	XMPP	CoAP	MQTT
Version	1.1	RFC 7622	RFC 7252	3.1.1
Standard status	Version 1.1, OASIS (2009)	IETF	IETF	Version 3.1.1, ISO/IEC 20922 (2016), OASIS (2014)
Type	Service Oriented protocol	Message Oriented protocol	Resource Oriented protocol	Message Oriented protocol
Transport	TCP & UDP	TCP	UDP (TCP extension planned)	TCP
Synchronous Communication	Yes (Service Invocation)	Near-real time	Yes (Request/response, via HTTP methods)	No
Asynchronous Communication	Yes (Publish/Subscribe to Service - WS-Eventing)	Yes (Publish/Subscribe)	Yes (Observe Resource - RFC 7641)	Yes (Publish/Subscribe to Topic)
Discovery	Yes (WS-Discovery)	Yes (XEP-0030)	Yes (RFC 5785 / RFC 6990)	No
QoS	Not integrated	Not integrated	Elementary support	Yes (3 modes)
Security	Payload encryption, WS-Security, TLS, IPsec, 802.15.4	SASL, TLS, Non-native end-to-end encryption	Payload encryption, DTLS, IPsec, 802.15.4	Payload encryption, TLS, IPsec, 802.15.4

Table 2.2: Feature comparison of the main IoT protocols

It is now time to explain why MQTT was chosen as an application layer protocol for this thesis. Firstly, based on the above comparison table, it is easy to see that MQTT offers greater QoS options against the other protocols, which makes it perfect for use on unreliable networks. On top of that, it is based on TCP instead of the unreliable UDP transport protocol, making it an even more reliable choice. One of the disadvantages of MQTT is that it has no discovery capabilities, but that is not a problem in this case, mainly because the scenarios of use for WSN nodes in IoT assume that all the properties (the sensors that are available and their names/topics) are already known and therefore there is no need for discovery. The other drawback of MQTT is that it is not a synchronous communication protocol. Once again, this is not a problem in our case, as there is no special need for synchronous communications between the WSN nodes and it is eventually counterbalanced by the large options of QoS. Finally, MQTT is the second most popular (after HTTP) IoT messaging protocol used today for IoT applications according to a recent survey (IoT Developer Survey 2016 [53]) done by the Eclipse IoT Working Group, IEEE IoT and Agile IoT on April of 2016 and this is an extra reason why MQTT was chosen.

3. The MQTT protocol

MQTT (formerly known as Message Queue Telemetry Transport) is an Internet of Things connectivity protocol, that runs on top of the TCP protocol, developed by IBM for lightweight Machine-to-Machine communications. It implements a publish/subscribe interaction model where the client devices do not need to impulsively request for updates; thus, reducing in that way the drain on resources and on power on the IoT nodes and making it optimal for use on high-latency or unreliable networks. For the same reason it is also used on mobile phone applications such as Facebook Messenger [37]. In 2013 IBM submitted MQTT v.3.1 to the OASIS specification body [27] and successfully completed the standardization of MQTT. Later, on 2014, the version 3.1.1 of MQTT was also approved [6] as a standard by OASIS. It is also standardized as ISO/IEC 20922 [41].

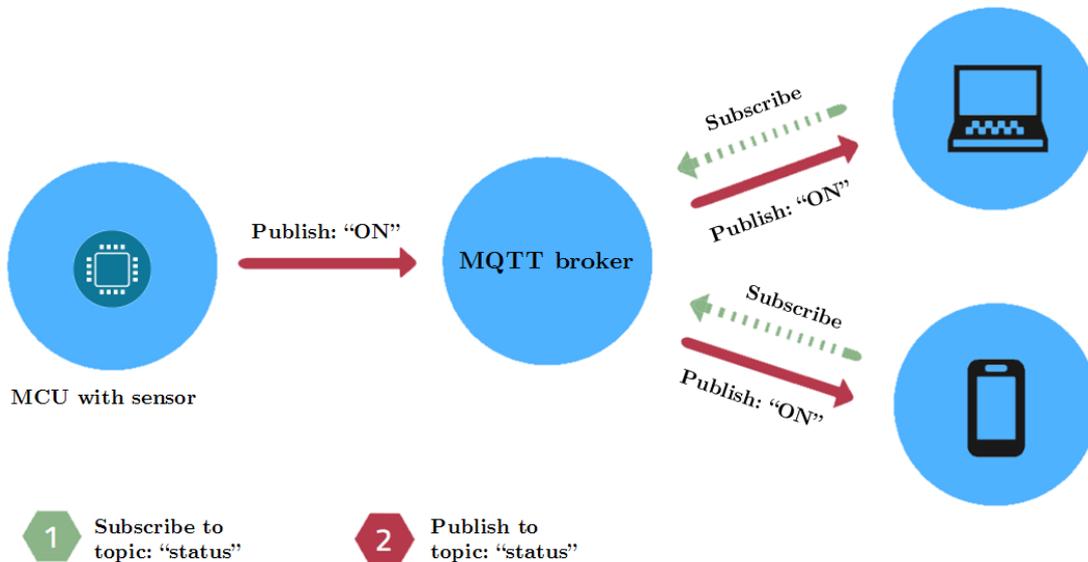


Figure 3.1: A simple MQTT publish/subscribe example

MQTT protocol follows the server/client schema with the server in this particular case to be called broker. The clients do not communicate directly with each other and all the messages (publishes) travel through the broker. Every message, except from its text (from now on referred to as payload), has a topic and each client can subscribe to various topics. Topics are organized in a hierarchical manner (called topic levels), with the form of file paths such as in a computer's file system; e.g. "home/bedroom/light/status". The broker receives the publish message from a client and is then responsible for relaying the message to every other client that has subscribed to this particular topic. It is easy to understand that MQTT was designed for one-to-many and many-to-many communications and with the assumption that a pre-defined relationship between participating nodes exists, because there are no discovery or content

negotiation mechanisms in the protocol. On the figure above (figure 2.3) a simple publish/subscribe example is depicted. MQTT has five basic methods/functions described by its specification, and could also have more, depending on the broker's implementation; they are: i) Connect, ii) Disconnect, iii) Subscribe, iv) Unsubscribe and v) Publish.

Another important feature of the MQTT protocol is the support of three different levels of QoS (Quality of Service). The term "QoS level", is, in fact, a form of agreement between sender and receiver of a message regarding the guarantees of delivering a message. MQTT's QoS levels are the following: i) level 0 – at most once (or better described as "fire and forge"), ii) level 1 – at least once (or "deliver at least once") and iii) level 2 – exactly once (or "deliver exactly once"). The assurance of the QoS levels greater than zero is achieved with the use of ACK (acknowledge) packets between the publisher and the broker.

Designed as a lightweight protocol for use on constrained devices, MQTT has some additional features towards that target. Namely, it has support for persistent sessions and message queuing for single clients, and the ability to retain messages on selected topics. The first feature is really helpful on unreliable networks where clients get often disconnected from the broker, and also saves a lot of resources on the client because there is no need to re-subscribe to all the topics after a reconnection. When a persistent session is configured, during the first connection of a client to the broker, it is ensured that all the subscriptions to the topics done by that client will be retained if the client gets unexpectedly disconnected from the broker. Additionally, all the messages that were published on the retained topics, with QoS greater than zero, will be sent to the client after it gets reconnected. The ability of the broker to retain the last message (together with its QoS) on a selected topic on the other hand is useful for newly connected clients because they are immediately updated about the status and they do not need to wait for the publishing client to send the next status update.

Finally, MQTT supports Last Will and Testament (LWT) messages and a configurable Keep Alive time interval for client-broker connections. The LWT feature is used in order to notify other clients about an ungracefully disconnected client by sending them a pre-defined (during the client's initial connection to the broker) message to a pre-defined topic (also QoS and retained flag could be pre-defined). Secondly, the configurable Keep Alive time interval is the longest possible period of time, which broker and client can endure without sending a message, and is really important for mobile networks due to their peculiar handling of TCP packets [42].

3.1. Security on MQTT

On account of the aforementioned difficulties encountered on implementing security mechanisms for the Transport Layer and the immaturity of the concept security mechanism for the Network Layer of constrained devices, our work will focus on establishing security on the Application Layer, as an alternative option to IEEE 802.15.4 security. At this section, all the security options available for MQTT (the Application Layer protocol that this thesis will focus on) will be described and commented.

3.1.1. Authentication on MQTT

“Authentication is the act of confirming the truth of an attribute of a single piece of data or entity”

When it comes to authentication in MQTT, the protocol provides a simple authentication through username and password fields in the Connect packet. Therefore, a client has the ability to send a username and a password when connecting to an MQTT broker, thus authenticating itself. The specifications also state that a username without password is possible in case identification is only needed instead of authentication for the client. The above option sends the username and password as plaintext to the broker and as a result it is not safe to use it without some other form of encryption of the sent packets, such as transport encryption (TLS).

An additional authentication method is by the use of the unique client identifier that every MQTT client registers at the broker at connection time. The client id can be up to 65535 characters and it is commonly given the value of the MAC address of the device or its serial number. The use of the simple username and password combined with this unique client id provides a good enough authentication method for closed systems such as a MQTT based implementation for home automation without internet access.

3.1.2. Authorization on MQTT

“Authorization is the function of specifying access rights to a certain resource”

A MQTT client can do two things after it has been connected to a broker; it can publish messages, and it can subscribe to topics. In the case when no authorization is performed, all connected clients could publish and subscribe to all kinds of topics.

Authorization on MQTT can be ensured by using an Access Control List (ACL) on the broker side. ACL is a list of permissions that specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects [30]. In the case of MQTT, ACL contains all the combinations of usernames and passwords and in what topics they have publish

and/or subscribe access. For example, when a client publishes to a topic it has no permission for, the broker has the following two options: a) It can disconnect the client, because publishing to a restricted topic is disallowed or b) It can acknowledge the publish to the client but decide not to send the published message to the subscribers. Additionally, extra broker authorization can be implemented with the form of plugins (if supported by the broker), or with the form of an extra web service running on the same machine as the broker.

3.1.3. Alternative to using TLS

Because, as it has already been argued, TLS is not feasible on constrained devices due to the insufficient resources available, another alternative must be found in order to ensure that the data sent to the broker are secured against third parties.

The best alternative to TLS is payload encryption. Payload encryption is the encryption of application specific data, typically the MQTT Publish packet's payload, the LWT data and on more extreme cases the username and password fields on the Connect packet. All other MQTT packet's metadata stay intact and only the payload of the message is encrypted. The encryption can be done by any of the available encryption algorithms (symmetric or asymmetric) as long as there is support for constrained devices. In addition, authenticated encryption algorithms can also be used to furthermore ensure message integrity. For the payload encryption solution there are two possible encryption scenarios:

i) End-to-End (E2E) encryption

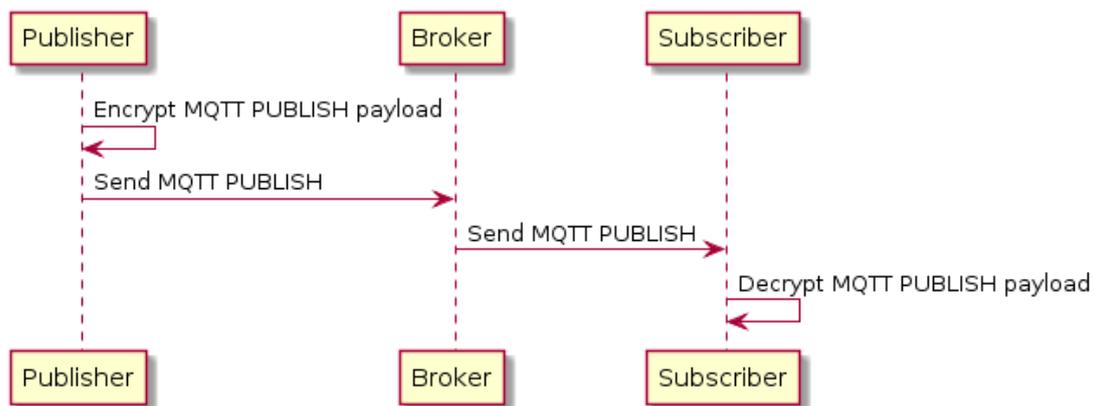


Figure 3.2: The End-to-End encryption scenario [31]

At this scenario, the payload is encrypted on the publisher's device and is decrypted on every subscriber that has the right key in order to decrypt, and is, therefore a trusted subscriber. The broker has no knowledge of the decryption key or of the payload's content. This approach ensures that the encrypted data stays encrypted all the time. The advantage of this scenario is that E2E encryption is broker independent and can be used on any topic by any MQTT

client. As a result, it is very good if, for some reason, the authentication and authorization mechanisms of the broker cannot be used.

ii) Client-to-Broker encryption

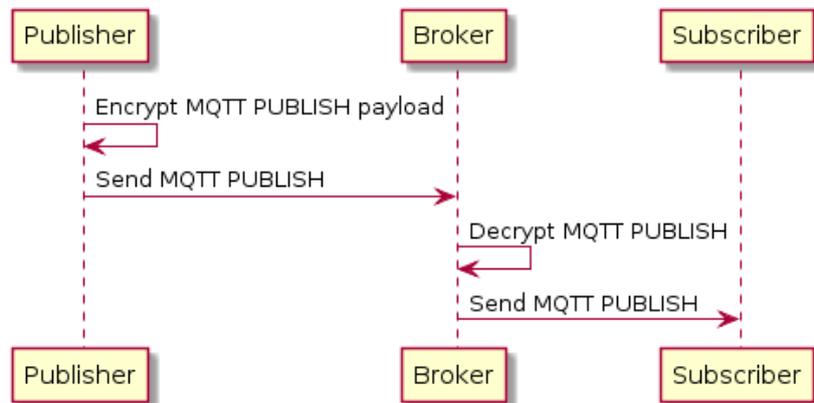


Figure 3.3: The Client-to-Broker encryption scenario [31]

This scenario called Client-to-Broker, ensures that the payload of the message is encrypted in the communication between one client (the subscriber) and the broker. This approach requires a custom-developed broker plugin that will decrypt the encrypted data on the broker side, and should only be used when trying to protect only the publishing side, while the subscribers are connected with a secure connection to the broker.

3.1.4. Other notes on MQTT security

On devices that are not restricted from constrained resources there are of course extra available options that can enhance the provided security. One commonly used method is the use of X509 client certificates together with TLS, in order to verify the identity of the client that tries to connect with the broker. On the other side, the broker also sends its X509 server certificates to all the clients that try to connect in order for them to validate its identity. Another option is the use of the OAuth 2.0 authorization framework that enables the access to a resource without the need of providing unencrypted credentials [32], and works in a similar way as the well-known Kerberos [46] authentication protocol.

4. Implementation

At this point, all the details about the tools utilized and the implementation process will be presented and discussed.

4.1. Tools Utilized

As all the technical details about the related technologies and the security mechanisms that accompany them have been thoroughly described in the previous sections, it is now time to present the tools that were used in order to achieve our target of fully implementing a secure and lightweight MQTT implementation for WSNs.

4.1.1. Contiki OS as an Operating System for IoT

The operating system is the foundation of the IoT devices as it provides the functions for the connectivity between them and all the necessary mechanisms needed for establishing security. An operating system designed for use in IoT devices must, in general, comply to the following requirements [33]: i) Low Random-Access Memory (RAM) footprint, ii) Low Read-Only Memory (ROM) footprint, iii) Support of multi-tasking, iv) Support of Power Management (PM) and v) Is a Soft real-time OS. Low RAM and ROM footprint are needed because the amount of RAM and ROM provided by that kind of devices is already limited to a small amount, in order to save energy. Also needed is the support of multi-tasking, because of the nature of the communications that take place between the IoT nodes, and are, in general, asynchronous. A good power management system is among the major aspects of a IoT OS, since it is imperative to conserve energy on battery powered devices by turning on and off peripherals such as flash memory, I/O, and sensors, but also by putting the MCU itself in different power modes. Finally, a soft real-time OS is needed because on that kind of real-time applications, latency and execution time cannot be guaranteed. Contiki OS [7] meets all the above requirements and it is the one we will use on this thesis. For reference, the below table shows a comparison of the most popular OS for IoT.

OS	Memory		Multi-tasking	PM	Real-time	
	RAM	ROM			Hard	Soft
Contiki	10kB	30kB	X	X	-	X
RIOT	1.5kB	5kB	X	X	X	X
TinyOS	1kB	4kB	X	X	-	X
freeRTOS	1kB	10kB	X	X	X	X

Table 4.1: OS requirements support comparison [33]

Contiki is an open source and community supported operating system designed for use in IoT and it is written in C language. It is currently supporting a large number of different MCUs and radios. As for protocols, it supports the commonly used TCP/IPv4 and IPv6 with the uIP (micro-IP) stack, and it also supports the 6LoWPAN stack and its own stack called RIME. The network stack implemented in Contiki is slightly different than the usual model typically adopted in TCP/IP. In-between the Physical and the Network layers, where, usually, the MAC is located, Contiki has three different layers: The Framer, the Radio Duty-Cycle (RDC) and the Medium Access Control (MAC). All these layers could be configured at compilation time by using their respective global variables.

Contiki also has threading capabilities with a thread system called Protothreads [34]. This type of threads does not use a call stack and therefore uses only two bytes of memory per thread. However, each thread is bound to one function and it only has permission to control its own execution. Contiki includes also libraries for a wide range of popular applications such as HTTP, Constrained Application Protocol (CoAP), UDP, and FTP servers, as well as other useful programs and tools, among them a MQTT client. As seen and in the above figure, a standard system with IPv6 networking capabilities running Contiki needs about 10 KB RAM and 30 KB ROM.

Built into Contiki is also the Cooja Network Simulator, a simulation environment that allows developers to see their applications run in large-scale networks or in extreme detail on fully emulated hardware devices. On the context of this thesis we will make full use of Cooja for prototype testing and performance assessment.

4.1.2. 6lbr as a 6LoWPAN Border Router

In order to deploy a complete MQTT system, with WSN nodes as clients and a broker running on Local Area Network (LAN), 6lbr was used as a Border Router (BR). A Border Router connects the 6LoWPAN devices to the local area IPv6 network and, therefore, the Internet, and is responsible for handling traffic to and from the IPv6 and 802.15.4. 6lbr [9] is an open source 6LoWPAN/RPL Border Router deployment-ready solution based on the Contiki OS. It can be deployed on low-cost, open source embedded hardware platforms like the RaspberryPi [35], the Econotag and the BeagleBone or even to a PC running Linux.

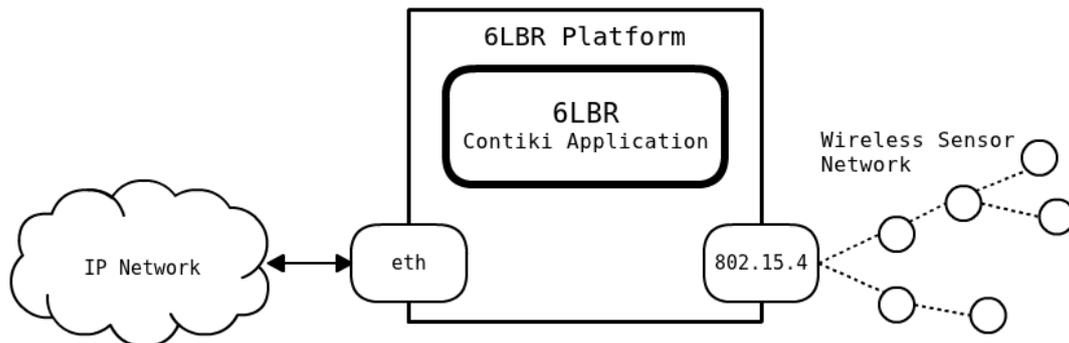


Figure 4.1: 6lbr border router diagram [9]

6lbr can be configured with different network architectures (such as Bridge, Router or Transparent Bridge) and has a variety of features; some of them are: network autoconfiguration, synchronization of 6LoWPAN WSNs with IP network and an enhanced webserver with configuration commands and monitoring capabilities. Moreover, on its latest version (version 1.4.x - still under development) it has full 802.15.4 Security Layer support

4.1.3. Mosquitto as a MQTT broker

Mosquitto [36] is an open source and cross-platform message broker that implements the MQTT protocol versions 3.1 and 3.1.1. It also offers all the security features mentioned on section 3.4 and is the MQTT broker that was used on the complete deployment of a MQTT system in this thesis.

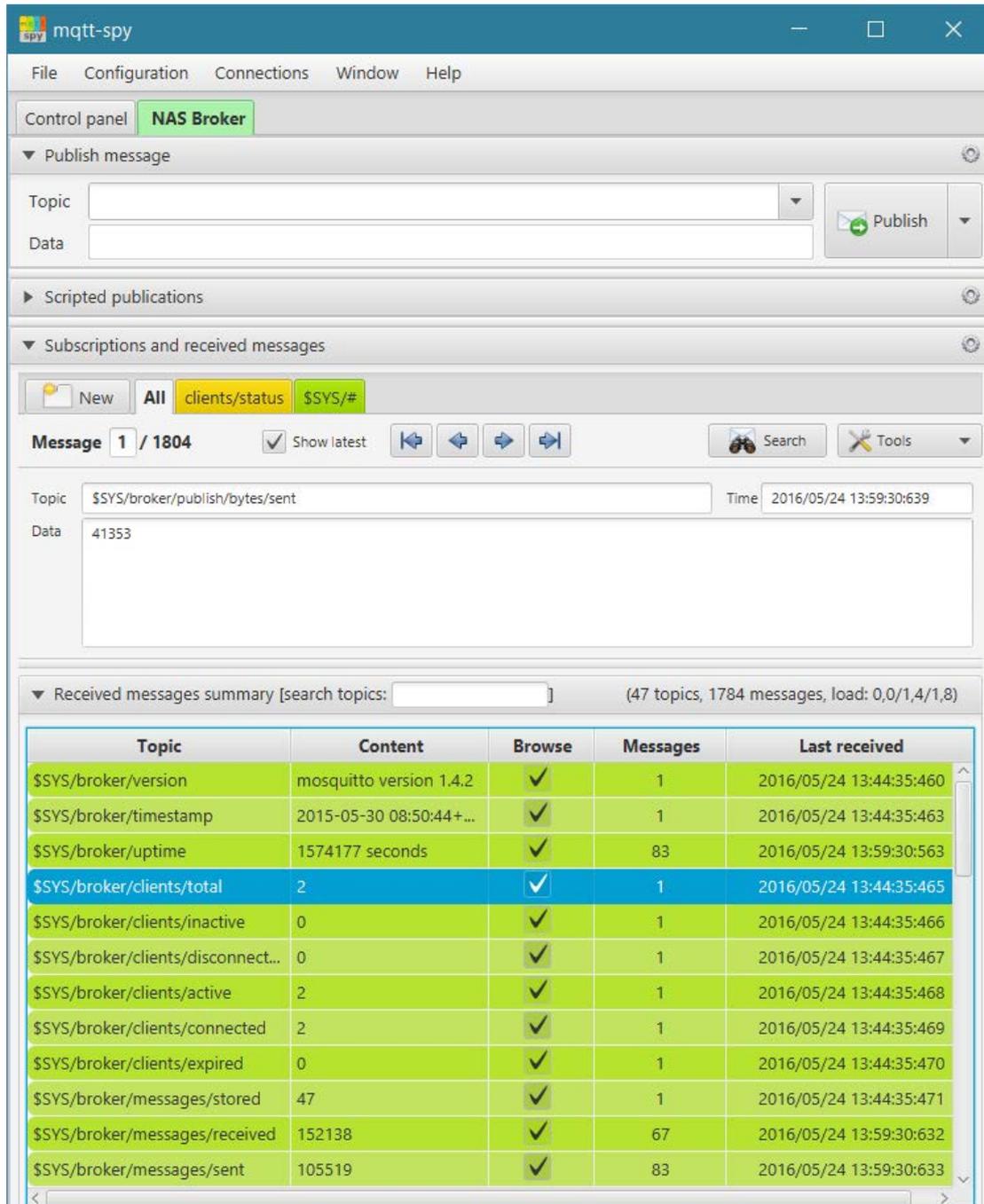


Figure 4.2: mqtt-spy (MQTT client) showing broker statistics

4.1.4. Zolertia Z1 as a hardware development platform



Figure 4.3: A Zolertia Z1 mote

When it comes to IoT, the second important decision that must be made, after the choice of OS, is the decision of which hardware platform to use. Today there are a lot of available options; ranging from ultra-low power and limited resources platforms (such as Zolertia Z1, Zolertia ReMote, Wismote, Skymote and many others), to medium power platforms (such as Arduino), or even more powerful platforms (such as Raspberry Pi). Except from power consumption and performance capabilities, extensibility in terms of enough I/O connections and popular interfaces support, as well as community support, must also be considered.

Zolertia Z1 [8] is a development platform that builds upon the second generation ultra-low power 16bit 16 MHz MSP430 RISC MCU [38]. The communication is managed by the Texas Instruments CC2420 radio transceiver which operates in the 2.4 GHz band. It is equipped with 8KB of RAM and a 92KB Flash memory, although only 56KB of it are usable without using the proper configuration. It also has a very well established support on many open source operating systems, including Contiki. Additionally, it comes with two built in sensors; the SHT11 temperature and humidity sensor, and the MMA7600Q accelerometer, and extensibility is ensured with: two standard (and two more available) connections designed especially for external sensors (called phidgets), an external connector with USB, Universal asynchronous receiver/transmitter (UART), SPI, and I2C support. Finally, the board can be powered in many ways, either from a battery pack (2xAA or 2xAAA batteries), by a coin cell battery (up to 3.6V), or from the USB connection.

4.2. General notes on implementation

To begin with, in the following table are presented the versions of the aforementioned tools and source codes that were used in the implementation phase of this thesis.

Tool name	Version	Notes
Contiki OS	3.0	Using commit 9bdb1f1 , 21 January 2016
6lbr	1.3.3	Using commit 3715b49 , 8 May 2015
Mosquitto	1.4.8	The latest version at that time
MQTT client	No version	Based on the MQTT example on Contiki's source code
msp430-gcc compiler	4.7.0	When using versions < 4.7.2 only 56KB of ROM can be used on Zolertia Z1
Development OS	Ubuntu 14.04 LTS	Comes as a VM image including Instant Contiki [39] 3.0 and Cooja simulator

Table 4.2: The versions of the tools used in implementation phase

In more detail, all the source code was developed and compiled inside a Virtual Machine (VM) running Ubuntu 14.04 LTS, using the msp430-gcc compiler on its 4.7.0 version. On the same VM, the Cooja simulator, on which all the testing and assessment was carried out, came preinstalled. It must be noted that is not always possible to use the latest commit of Contiki because it gets updated very often and many bugs occur from one version to the next. In order to avoid problems with unexpected bugs, the above commit on the master branch from 21 January 2016 was used for all the implementations. On the other hand, 6lbr does not get updated so often on its master branch (all the updates under development are done on the develop branch, leaving the master branch intact until a final release is ready) so the latest stable version, at that time (version 1.3.3), was used.

It must be also noted that there were two different evaluation environments configured and used during that phase: i) The virtual environment and ii) the real world environment. In the first environment, all the WSN motes programmed as MQTT clients were running inside the Cooja simulator on virtual Zolertia Z1 motes and were connected to a MQTT broker running on the VM's localhost with the use of Contiki's Native Border Router. On the second environment, the MQTT clients ran on real Zolertia Z1 motes and were connected to a MQTT broker running on a Windows machine connected to the LAN with the use of 6lbr as Border Router that was configured on a Raspberry Pi Model B. The first setup was used during code development

to find bugs and quickly test the written code, whereas the second setup was used for further testing the implementations on real world scenarios.

4.2.1. Testing setup topologies

On the below two figures the two different topologies for each one of the two environments that was used are depicted:

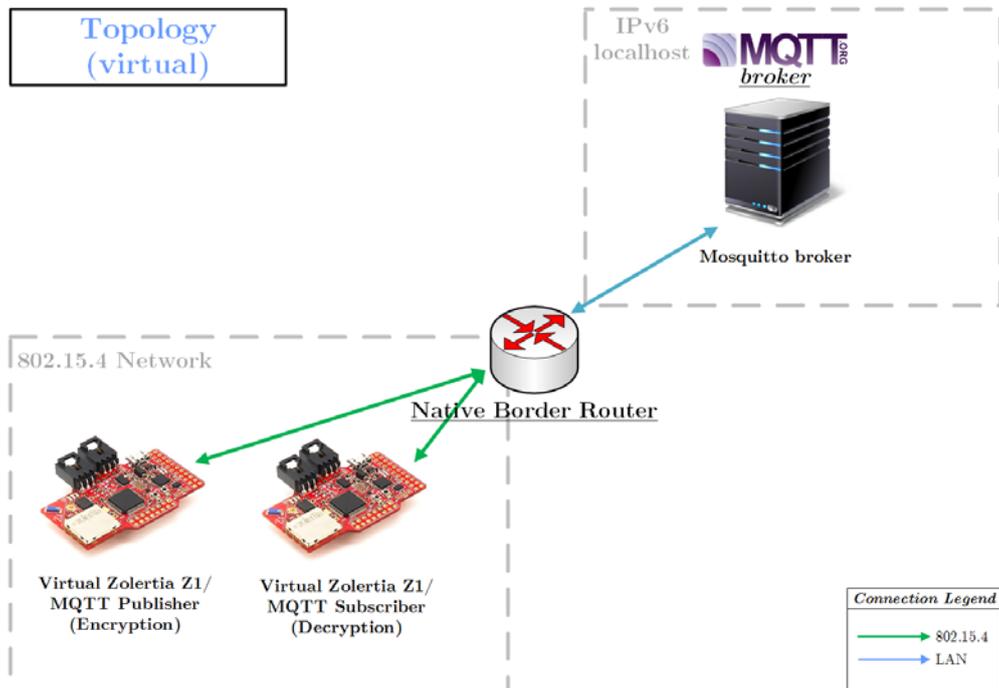


Figure 4.4: The topology of the virtual environment

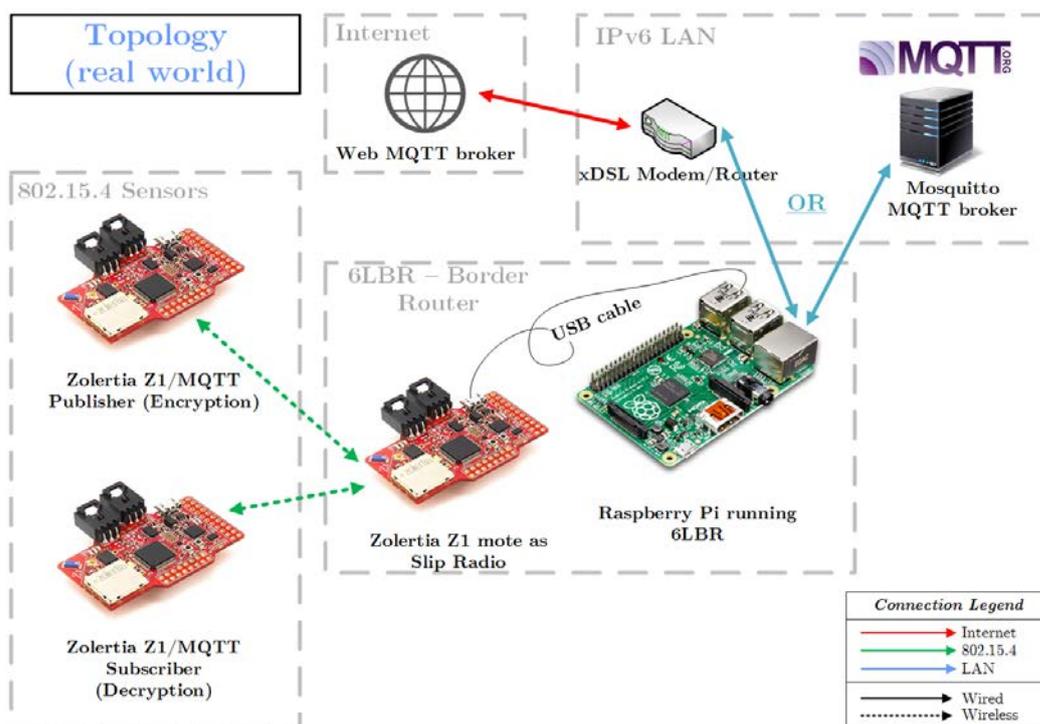


Figure 4.5: The topology of the real world environment



Figure 4.6: Real motes deployed during evaluation

One more thing that should be noted about the real world testing environment (which is also shown on the above figure), is that the motes were placed close to each other, in order to eliminate possible problems caused by low signal level because our only goal was to perform a performance evaluation that is independent of signal level.

4.2.2. Configuration of the WSN motes

Contiki has a lot of parameters that can be configured at compilation time by changing the respective global variables on the `project-conf.h` file of the compiled program. In this case the interest is towards the variables that configure the extra three layers (RDC, MAC and Framer) on the network stack as described on the section 4.1. More specifically, in our setup, the MAC layer driver was set to the Contiki's default one, called "nullmac_driver", and is equivalent to not having a MAC mechanism enabled and the RDC layer driver was set to the "contikimac_driver" [40] which is a RDC mechanism that tries to keep the radio transceiver in off position in order to save as much energy as possible. With the above configuration we try to achieve the lower power consumption together with minimum additional overhead on ROM and RAM of the mote.

4.2.3. 6lbr configuration

For use in the real world testing environment, 6lbr was installed and configured as a RPL router on a Raspberry Pi Model B. No changes were done on the configuration of the WSN network part, but on the Ethernet network part it was configured to give addresses on the IPv6 subnet that was used in

the LAN. The IPv6 LAN was maintained by a xDSL router with IPv6, Router Advertisement (RA) messages support and unique local addressing (ULA) capabilities. Then, a Zolertia Z1 was programmed with the slip-radio code from 6lbr (because the one on the Contiki's source code did not have support for the Zolertia Z1) and was connected via USB on the Raspberry Pi. Finally, the Raspberry Pi was connected to the LAN with an Ethernet cable.



Figure 4.7: 6lbr deployed on Raspberry Pi

4.2.4. MQTT client source code

The source code used for the MQTT client implementation is based on the MQTT client example (found on `mqtt-demo.c` file) of the Contiki's source code on GitHub, and uses the, also built in, MQTT library. However, the source code was heavily modified with the removal of redundant lines of code (including defines, duplicate variables, not used functions and includes) and the reduction on size of many buffers in order to gain as much extra free space as possible on the ROM and RAM that will be used for the additional security implementations. Additionally, the MQTT example had support for the IBM Quickstart platform (currently called IBM Watson IoT platform) that was completely removed for the same reason. The resulting source code with some other additions (such as an extra connection status led support) was used as a foundation for implementing the security mechanisms for the MQTT client.

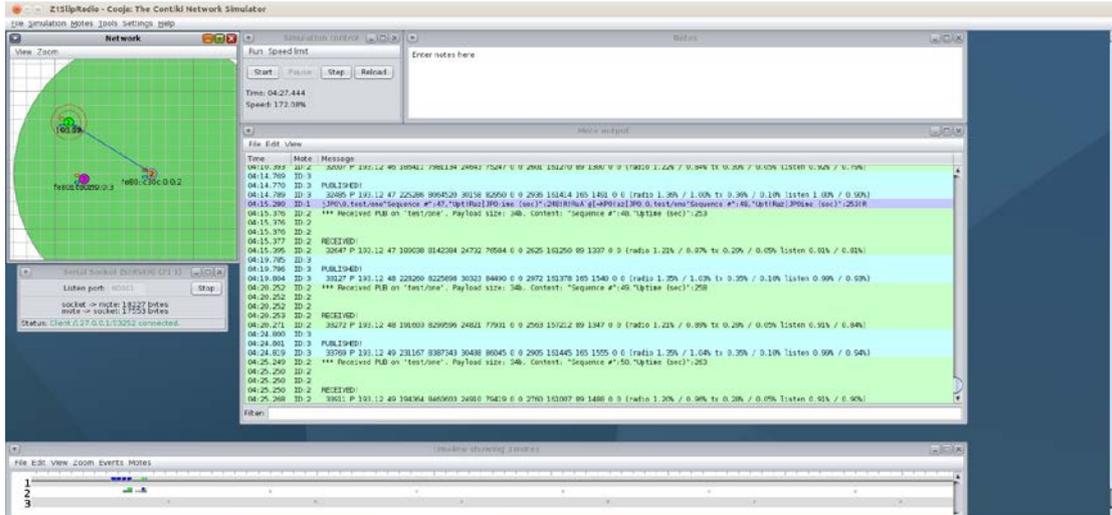


Figure 4.8: The simple MQTT client running on Cooja

4.3. Secure and lightweight MQTT implementations

On the following sections all the secure and lightweight MQTT client implementations that were developed in the context of this thesis will be presented.

4.3.1. Option 1 – Payload encryption with AES

The first implementation ensures the security of the transferred messages with the encryption of the payload on the MQTT packet using symmetric key end-to-end (i.e. client-to-client) encryption, and, more specifically, using AES with a 128-bit key. Instead of using the Contiki’s built in AES implementation a Texas Instruments AES implementation in C language for the MSP430 MCU [51] was used. The above choice was justified by the fact that the Contiki’s built in AES implementation includes only the encryption function (i.e. no decryption function), since it is only used for signing in the CCM* implementation. This implementation encrypts messages with only one block size length (the block size in AES is 16 bytes) and the message must be manually padded if it has smaller length. Additionally, support for larger payload sizes has been added with the form of multiple consecutive encryptions/decryptions of 16 bytes, as it is done in the ECB mode of operation. However, this mode is not the safest one and it is vulnerable to replay attacks because every same block is encrypted in the same ciphertext.

4.3.2. Option 2 – Payload encryption with AES-CBC

This implementation is an extension of the previous one, enabling the encryption of messages that have lengths different than one block size with the use of AES in CBC mode of operation. For this, the AES-CBC implementation from ContikiSec [52] was used. It must be highlighted that even though there

is no specific limit on the length of the messages to be encrypted, this implementation was restricted to lengths up to four block sizes (i.e. 64 bytes) because of the limited amount of RAM. This size can of course be easily changed for use in other platforms as it is defined in the start of the source code.

4.3.3. Option 3 – Payload authenticated encryption with AES-OCB

The third implementation comes with an important addition to the payload encryption; that is, authentication. In this implementation the AES-OCB authenticated encryption mode of operation was used in order to furthermore ensure the integrity of the transferred messages. The AES-OCB implementation from ContikiSec [52] was used. Together with the encrypted payload the tag is also appended on the message to be transferred, and is used from the receiving MQTT client for validation of the encrypted message. Again, due to the limited RAM on the platform we used, the messages to be sent were restricted to the length of two block sizes. Another note that must be made for this implementation, is that due to the size of the compiled program, it was unable to include both the encryption and the decryption functions on the same mote. As a result, for testing this implementation two motes are needed; one publishing the messages, and another one, subscribed to the topic, that receives and decrypts the messages.

4.3.4. Option 4 – Link layer encryption with CCM*

Finally, the last secure MQTT client implementation has a different approach than the three previous. For this implementation, Link Layer encryption (link layer security – LLSec) using AES-CCM-128 was used instead of payload encryption. This ensures the node-to-node security of all the transmitted package (including the topic, the client id etc.). This security mechanism is included in the 802.15.4 as described before, so the only change on the source code that made was the definitions of the operating system's parameters that enable this mechanism. It must be noted that even though this solution provides complete security, it has a simple drawback; if a group key (the same key for all nodes) is used then all the nodes would have access on the data. Another significant disadvantage over the payload encryption is that every node that receives a publish packet must decrypt it first in order to find out what the topic is and then discard it if no subscription is done on this topic, thus consuming extra resources. Finally, LLSec could be used together with MQTT's simple username/password authentication capabilities in order to provide a complete confidentiality and authenticity solution.

4.3.5. Implementation diagrams

The two following diagrams show the publish and receive subscribed publish actions according to the four implementations referred to above.

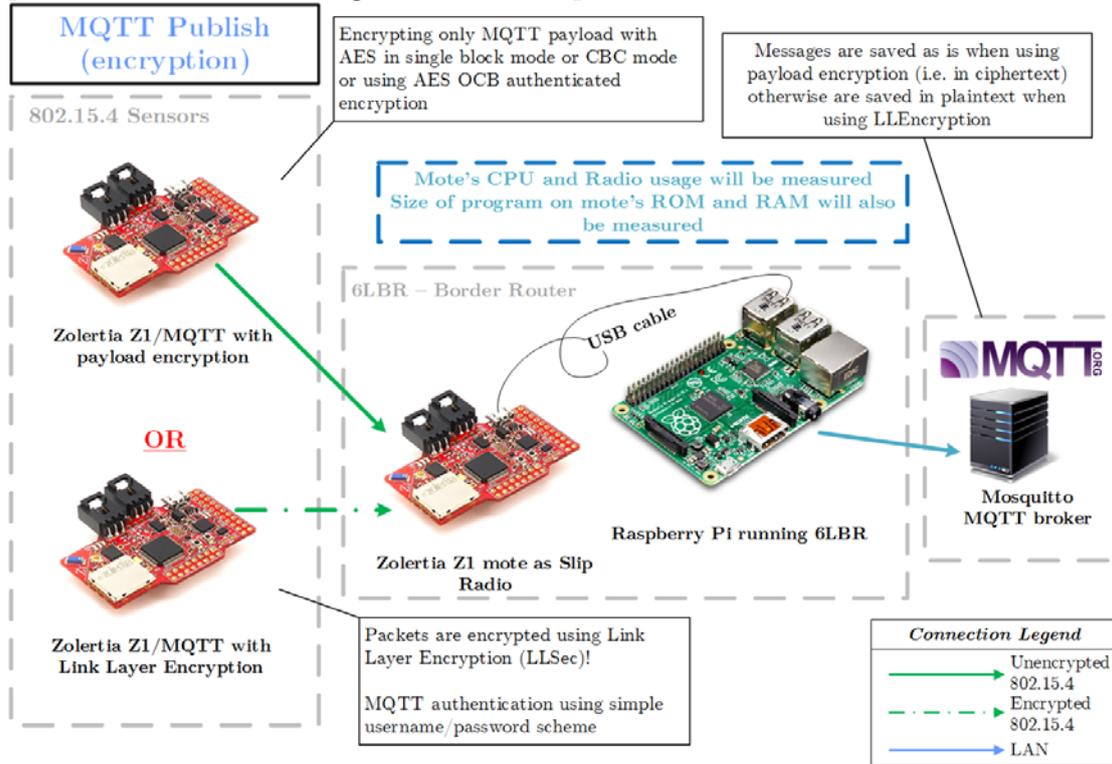


Figure 4.9: The MQTT publish action diagram

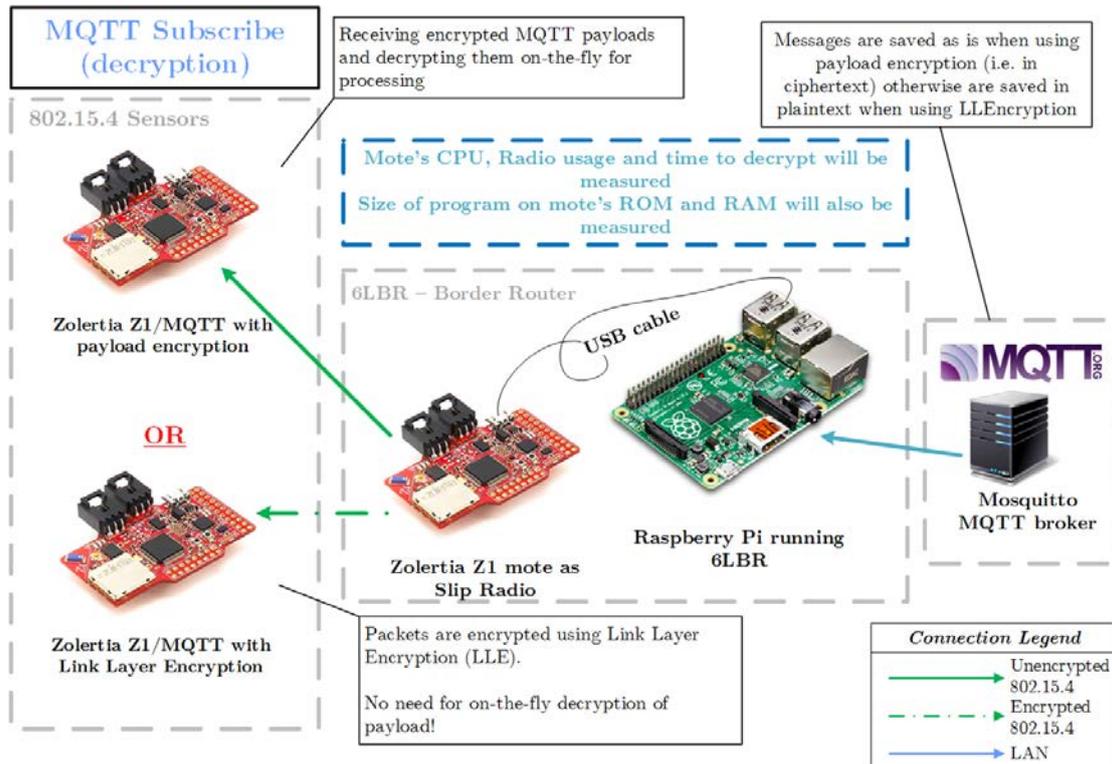


Figure 4.10: The publish to a subscriber client action diagram

5. Performance Assessment/Evaluation

In this chapter, the performance evaluation of the four different security mechanisms that were developed for this thesis will be presented, along with other useful information about them. In order to have a reference about the performance of every security mechanism, first will be measured the performance of the simple MQTT client (called option 0 from now on).

Before we begin presenting the performance evaluation results let us first describe the way the performance measurements were carried out. Firstly, it must be noted, that during the performance evaluation, two different (both virtual or real, according to the environment) motes were used, one as a publisher (encryption) and another one as a subscriber (decryption) and a number of 50 MQTT messages were published for every performance evaluation. For measuring the MCU's and the radio's power consumptions a built in power consumption software measuring tool of Contiki, called Powertrace was used. Energest is another built in software tool that uses wraparound macros to count the number of CPU timer ticks in each power state (high and low power CPU modes, radio RX and TX) the platform wants to keep track. Powertrace uses Energest along with a periodic difference of the CPU timer ticks to get average power over a shorter period of time, or for particular network modes. It then prints out a serial output with the collected data every time its method is called, or a specified amount of time. More specifically, the energy consumption and the radio duty cycle can be calculated using the following two formulas:

$$\text{Energy [mW]} = \frac{\text{Energest_Value} \times \text{current [mA]} \times \text{voltage [V]}}{\text{RTIMER_ARCH_SECOND} \times \text{Time_Duration}} \quad (1)$$

Where “Energest_Value” is the periodic value printed out by Energest, “RTIMER_ARCH_SECOND” is the number of ticks performed by the internal CPU timer per second, that is 32768 in this case and finally, “Time_Duration” is the time in seconds from the previous Energest measurement.

$$\text{Radio duty cycle [\%]} = \frac{\text{Energest_TX} + \text{Energest_RX}}{\text{Energest_CPU} + \text{Energest_LMP}} \quad (2)$$

Where the “Energest_XX” is the corresponding value printed out by Energest.

On the other hand, the message latency, more specifically the time between the publish (including the encryption, if done) of a message and its processing from the receiving end, was measured with two different methods according to the testing environment. On the virtual testing environment, the message latency was easily and more accurately calculated by measuring the time difference (on the simulator’s serial prints timestamps) between two serial debug prints; one after the message was successfully published by the publisher, and the other, after the message was received and decrypted (if it was encrypted) from the subscribed client. On the real world testing environment, the latency measurement was not such an easy task because we cannot calculate the time difference between the two aforementioned debug print, since there are no (synchronized) timestamps. In order to measure the message latency, we implemented the following simple mechanism:

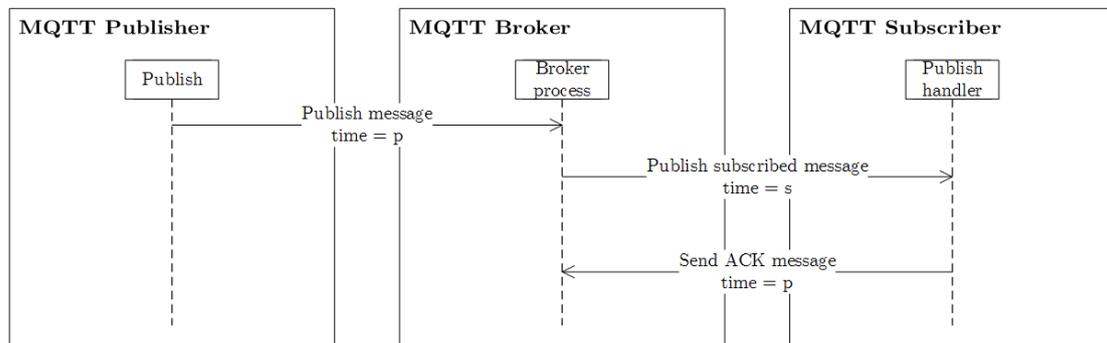


Figure 5.1: The ACK mechanism for latency measurements

This is similar to the previous method (for the virtual environment) but instead of a debug serial print on the subscriber’s end, an ACK publish message is used. That is to say, that after the subscriber receives the message and decrypts it (if it is encrypted), then it performs a MQTT publish on another topic (“clients/ack” for example). If we assume that every publish (and only the publish; after the message was encrypted/decrypted) takes time p and every subscribed publish takes time s then, one can readily see that the message latency is calculated by performing the addition of s and p and then, adding on the previous sum, the time needed for the encryption on the publishing mote. The sum of s and p could easily be calculated from the time difference on the timestamps of the messages that arrive at the broker, while the time to encrypt on the publishing mote could be calculated with the use of the internal runtime clock. As the maximum possible accuracy is needed, the MQTT broker was running on the same machine (i.e. on localhost), together with a MQTT client that received all the above messages and then the time difference of their timestamps was calculated.

Finally, the size of the compiled program was easily obtained with the use of the “size” command that provides information about the usage of different parts of the ROM (called “text” and “data”) and the usage of RAM (called “bss”).

5.1. Power specifications of Zolertia Z1

At this chapter the power specifications of Zolertia Z1, the hardware platform that was used, will be presented. The following table comes from the Zolertia Z1 datasheet [38] and displays the power (voltage and current) consumptions of the most basic power modes, such as, Active Mode (AM) and standby mode or also called Low Power Mode (LMP). The power specifications of the CC2420 radio IC are also presented.

IC	Voltage Range	Current		Mode
MSP430f2617	1.8V to 3.6V	0	μA	OFF Mode
		0,50		Standby Mode
		1	mA	Active Mode @1MHz
		< 10		Active Mode @16MHz
CC2420	2.1V to 3.6V	<1	μA	OFF Mode
		20		Power Down
		426		IDLE Mode
		18,8	mA	RX Mode
		17,4		TX Mode @ 0dBm

Table 5.1: Power specifications of Zolertia Z1

Although almost all of the above data are very useful, it is not clearly stated the current consumption of the MCU when running at frequencies lower than 16MHz. That is needed in our case because Contiki is running at 8MHz. For this information someone can refer to the detailed datasheet of the MSP430f261x MCU [54]. On the detailed datasheet the current consumption per 1MHz is displayed and the total current consumption according to the system's frequency can be easily calculated with the following formula:

$$I(\text{AM}) = I(\text{AM}) [1\text{MHz}] \times f(\text{system}) [\text{MHz}] \quad (3)$$

In our case the rated typical current at 1MHz when the program is running from flash memory and with a voltage of 3V is 515 μA . Thus the total rated current for operation at 8MHz is calculated at 4.12 mA. Also, the maximum rated current for operation at 8MHz is calculated at 4.48 mA. For the power consumption calculation, the average of these two values (that is 4.3 mA) will be used.

5.2. Evaluation of option 0 (simple MQTT client)

This is the simplest implementation, as there is no encryption or other security mechanisms used and the performance assessment is only done for using it as a reference for the other mechanisms. For this, MQTT messages with payload length of 32 bytes was used and the following data are from the performance measurements on the real world environment.

The average power consumptions and the radio duty cycle are presented on the following tables:

Average power consumption (mW)					
	CPU	LPM	TX	RX	Total
Publisher	0,251648	0,001470	0,094525	0,680525	1,028170
Receiver	0,270352	0,001468	0,084002	0,678472	1,034295

Table 5.2: Option 0 - Average power consumptions

Average radio duty cycle (%)			
	TX	RX	Total
Publisher	0,18%	1,21%	1,39%
Receiver	0,16%	1,20%	1,36%

Table 5.3: Option 0 - Average radio duty cycle

At this point it should be reminded that the average radio duty cycle values are this low due to the fact that the RDC mechanism was enabled in order to save energy.

The average latency of the messages, on the other hand, was **568.44 ms**.

Finally, the size of the program on the hardware platform is presented on the table below.

Program size (bytes)			
	ROM		RAM
	text	data	bss
Publisher	53149	306	6860
Receiver	52993	306	6758

Table 5.4: Option 0 – Program size

A more detailed view of the power consumption is shown on the following graphs:

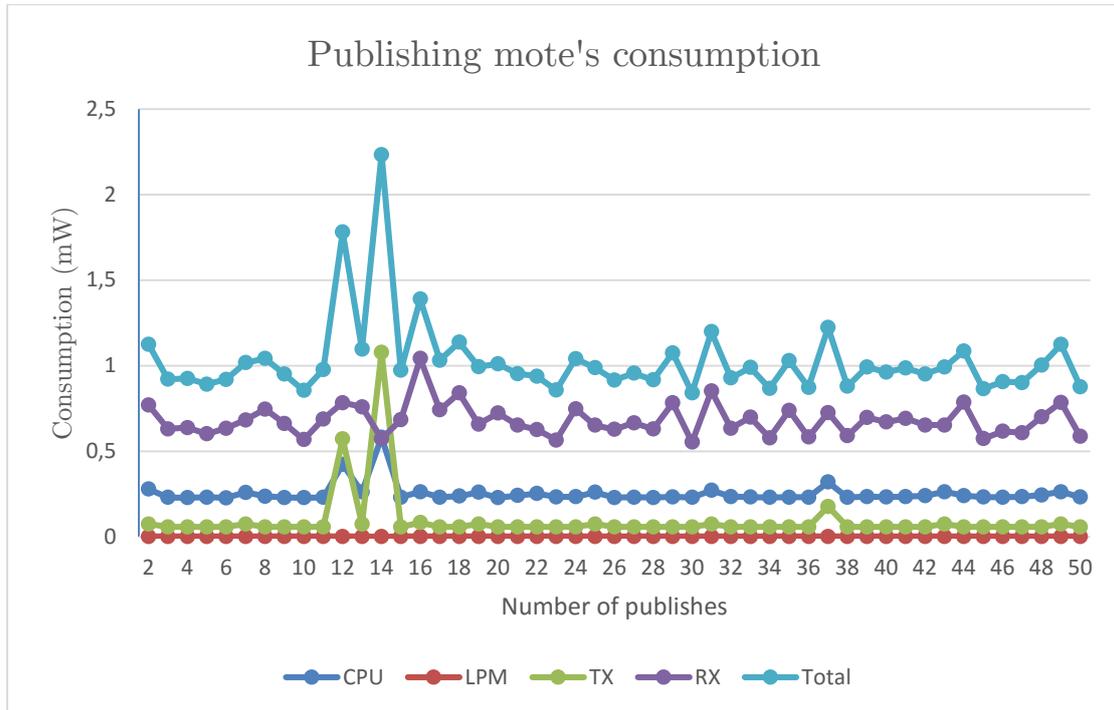


Figure 5.2: Option 0 – Publishing mote’s consumption graph

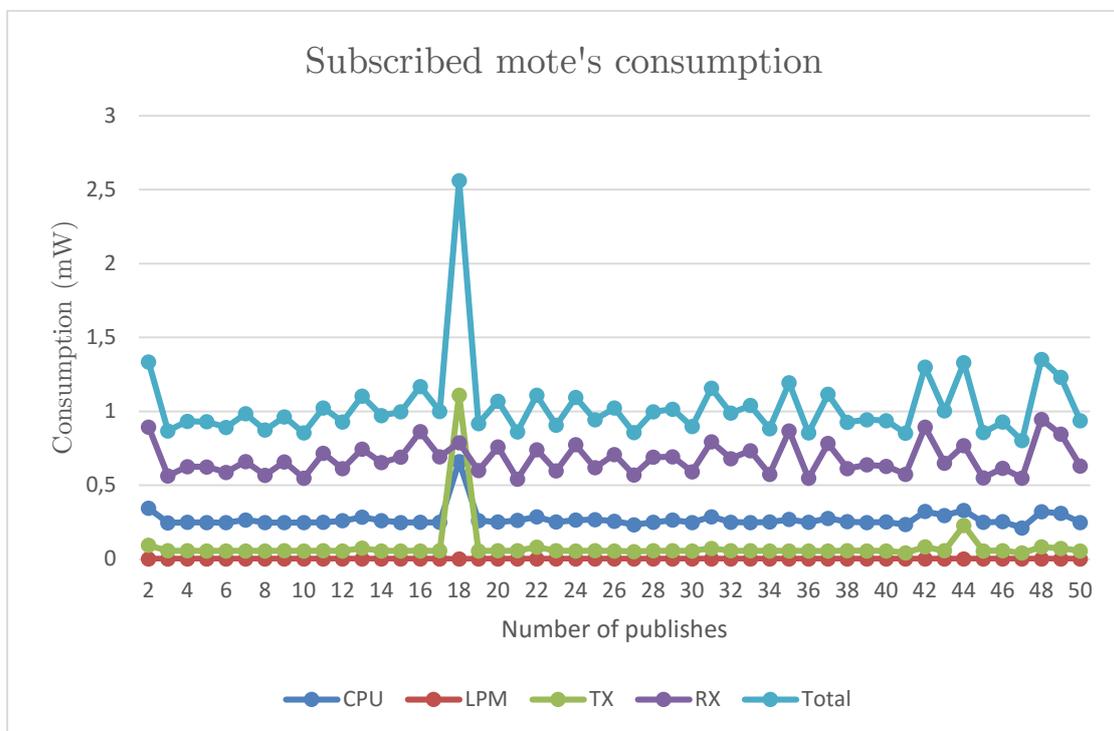


Figure 5.3: Option 0 – Subscribed mote’s consumption graph

5.3. Evaluation of option 1 (single block AES)

This option includes very basic payload encryption with the limitation of encrypting messages with payload size up to 16 bytes only due to the single block AES implementation used. It must be emphasized that this option could not be proposed as a full security solution because of the above limitation but on the other hand it has research interest as it could be used for comparison with the more complex encryption mechanisms used on the other options.

The average power consumptions and the radio duty cycle are presented on the following tables:

Average power consumption (mW)					
	CPU	LPM	TX	RX	Total
Publisher (encryption)	0,274662	0,001468	0,118635	0,658590	1,053357
Receiver (decryption)	0,300105	0,001465	0,083061	0,660250	1,044882

Table 5.5: Option 1 - Average power consumptions

Average radio duty cycle (%)			
	TX	RX	Total
Publisher	0,23%	1,17%	1,40%
Receiver	0,16%	1,17%	1,33%

Table 5.6: Option 1 - Average radio duty cycle

The average latency of the messages when using this mechanism was quietly larger than before due to the time needed for encryption and decryption of the messages and was **749.36 ms**. Moreover, the average time needed for the encryption and decryption of the message was calculated on the virtual evaluation environment to be about 10 ms.

Finally, the size of the program on the hardware platform is presented on the table below.

Program size (bytes)			
	ROM		RAM
	text	data	bss
Publisher	55035	322	6798
Receiver	55919	322	6768

Table 5.7: Option 1 – Program size

A more detailed view of the power consumption is shown on the following graphs:

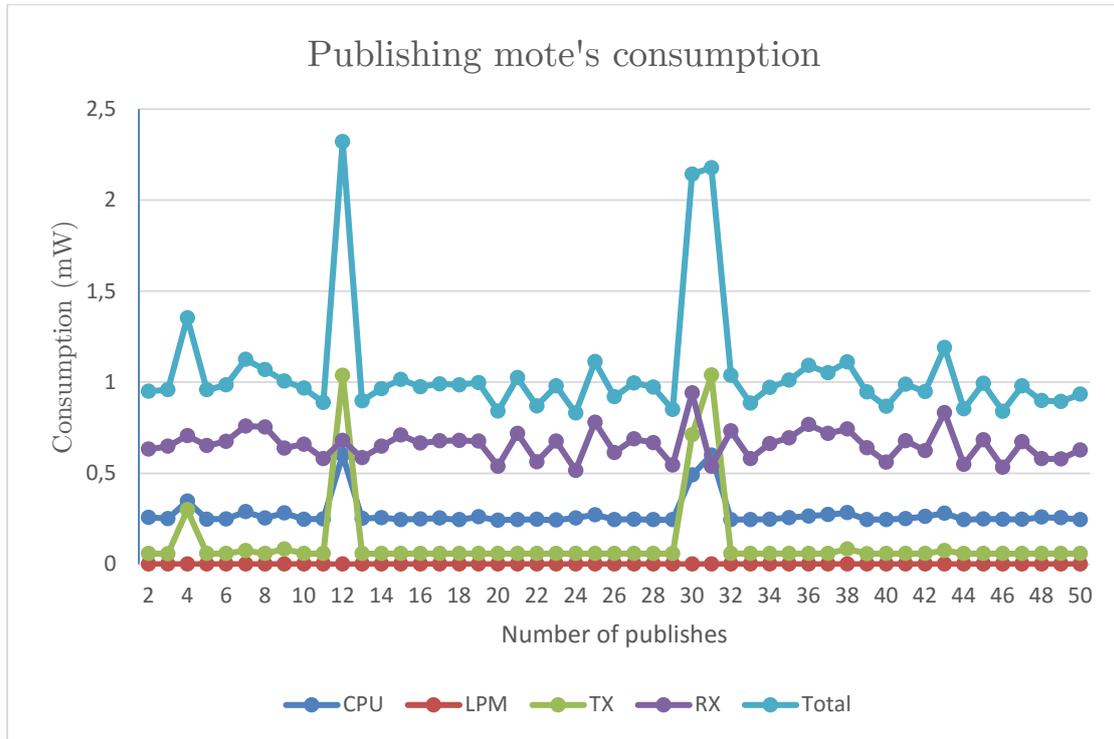


Figure 5.4: Option 1 – Publishing mote’s consumption graph

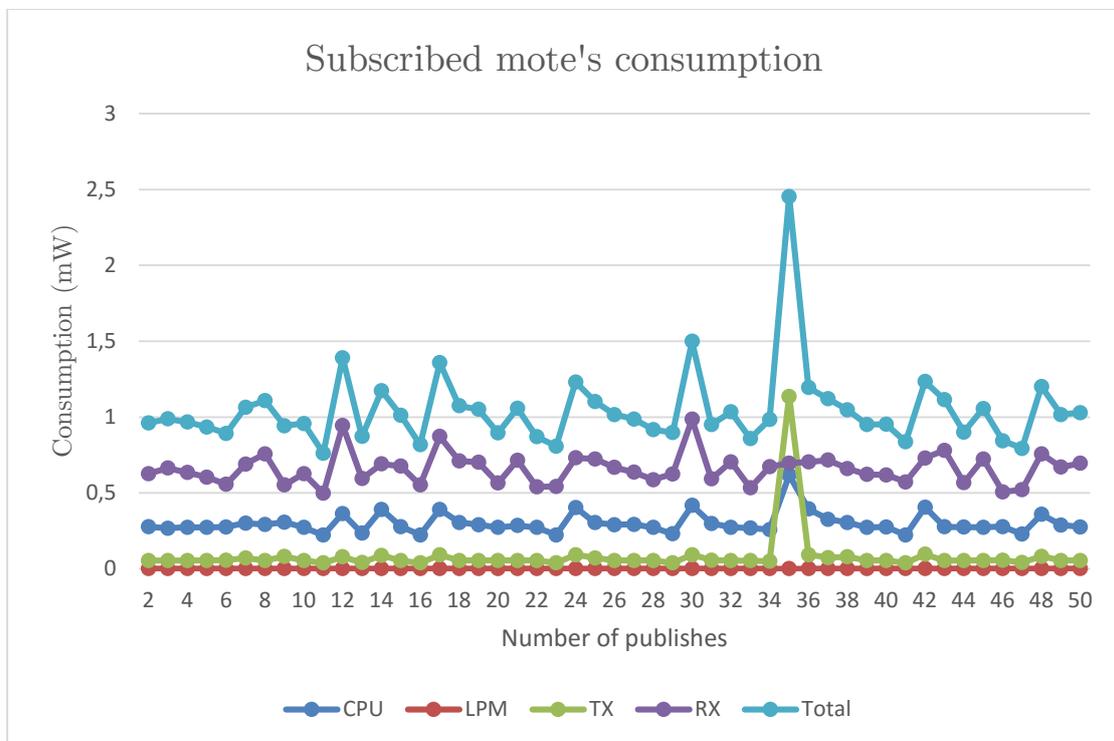


Figure 5.5: Option 1 – Subscribed mote’s consumption graph

5.4. Evaluation of option 2 (AES-CBC)

This option is the extension of option 1 and is a fully proposed security solution as it can ensure the confidentiality of messages with a variety of payload sizes using AES in CBC mode of operation. For the performance evaluation of this option, payload sizes of 32, 48 and 64 bytes were used.

The average power consumptions and the radio duty cycle for 48 bytes of payload are presented on the following tables:

Average power consumption (mW)					
	CPU	LPM	TX	RX	Total
Publisher (encryption)	0,372097	0,001456	0,195647	0,728457	1,297658
Receiver (decryption)	0,476729	0,001444	0,218100	0,912204	1,608479

Table 5.8: Option 2 - Average power consumptions

Average radio duty cycle (%)			
	TX	RX	Total
Publisher	0,37%	1,29%	1,67%
Receiver	0,42%	1,62%	2,04%

Table 5.9: Option 2 - Average radio duty cycle

When using such an advanced method of security a reasonable amount of latency is expected due to the execution time of the encryption and decryption functions. In this case the average message latency was, not significantly greater than in the simple AES, at just **764.88 ms**. The average radio duty cycle is also greater in this option as a result of the larger message payload size.

Finally, the size of the program which, as expected is larger, is presented on the table below.

Program size (bytes)			
	ROM		RAM
	text	data	bss
Publisher	55373	322	7134
Receiver	56747	322	7122

Table 5.10: Option 2 – Program size

A more detailed view of the power consumption is shown on the following graphs:

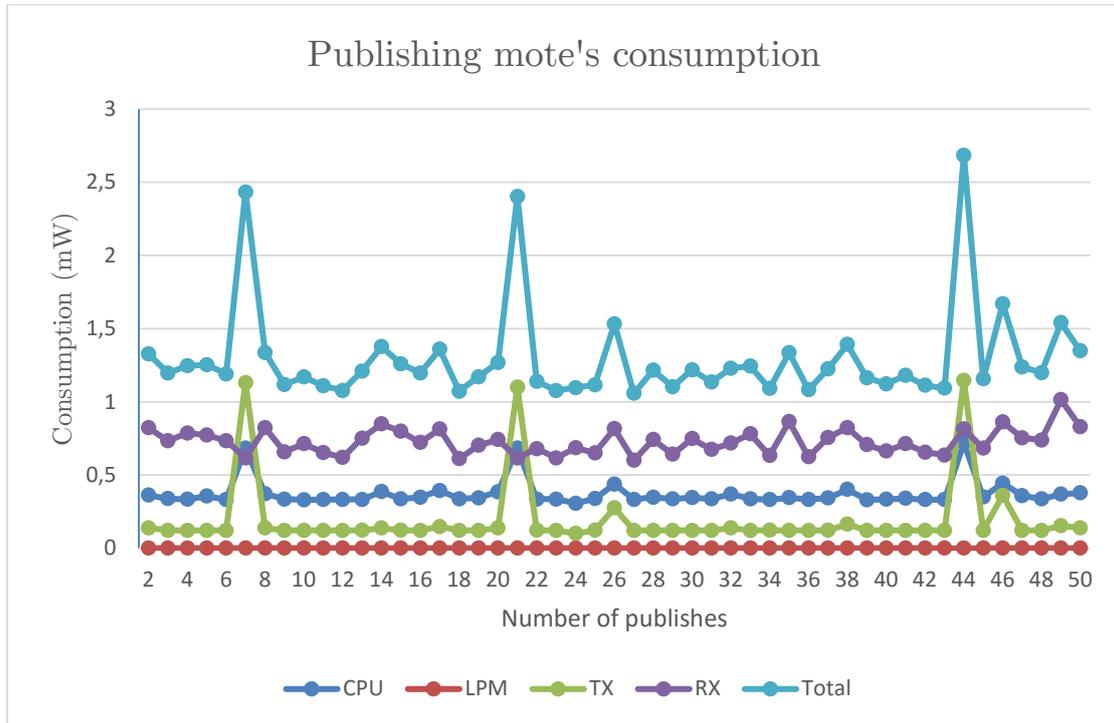


Figure 5.6: Option 2 – Publishing mote’s consumption graph

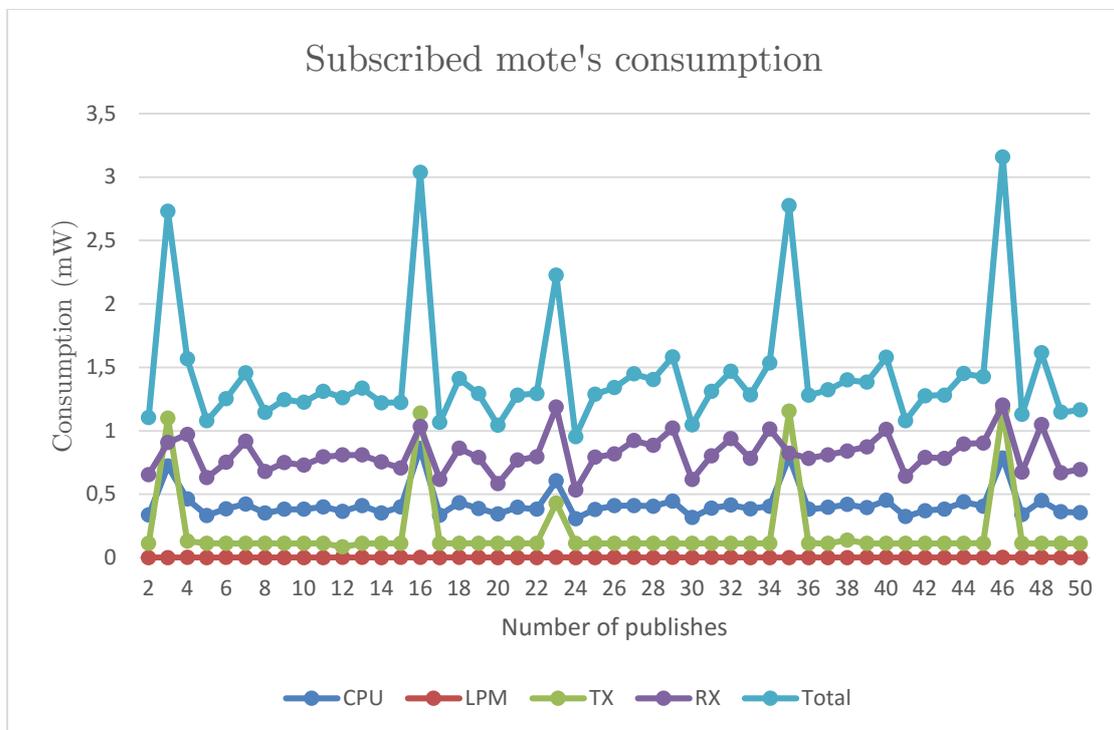


Figure 5.7: Option 2 – Subscribed mote’s consumption graph

The average radio duty cycle is also greater in this option as a result of the larger message payload that has an extra overhead of 16 bytes for the Tag. The average time needed just for the encryption and decryption of the message was calculated on the virtual evaluation environment to be about 33 ms.

Finally, the size of the program which, is also expected to be larger, is presented on the table below.

Program size (bytes)			
	ROM		RAM
	text	data	bss
Publisher	55959	338	7008
Receiver	56193	338	6848

Table 5.13: Option 3 – Program size

A more detailed view of the power consumption is shown on the following graphs:

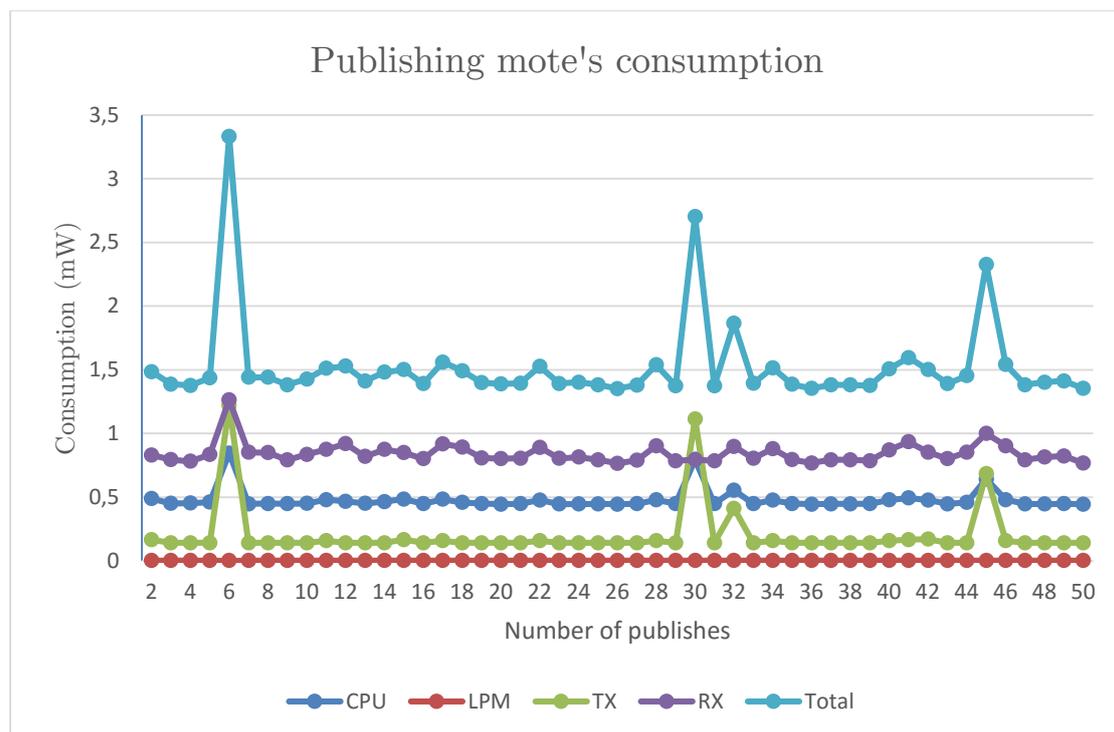


Figure 5.9: Option 3 – Publishing mote's consumption graph

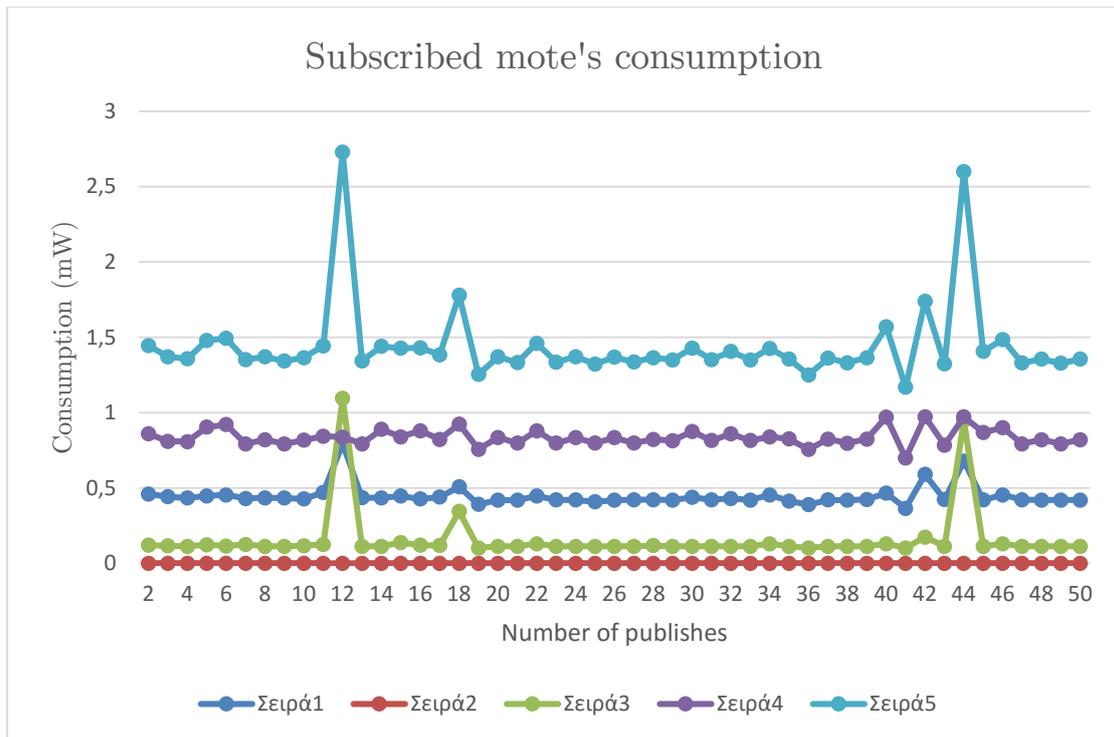


Figure 5.10: Option 3 – Subscribed mote's consumption graph

5.6. Evaluation of option 4 (Link Layer Security)

This security method is different from the previous ones because it is implemented on the Link Layer instead of the Application Layer. Additionally, it is built into Contiki and the only adjustment was to enable it from the configuration parameters of the OS. For the performance evaluation of this option, payload sizes of 32 and 48 were used.

The average power consumptions and the radio duty cycle for 48 bytes of payload are presented on the following tables:

Average power consumption (mW)					
	CPU	LPM	TX	RX	Total
Publisher (encryption)	0,266438	0,001469	0,128768	0,654116	1,050792
Receiver (decryption)	0,350747	0,001459	0,160960	0,742848	1,256015

Table 5.14: Option 4 - Average power consumptions

Average radio duty cycle (%)			
	TX	RX	Total
Publisher	0,25%	1,16%	1,41%
Receiver	0,31%	1,32%	1,63%

Table 5.15: Option 4 - Average radio duty cycle

In this case the average message latency was **1036.08 ms** which is significantly lower in comparison with AES-OCB. The main reason for this difference is probably the level on which the encryption takes place with the Link Layer to be more effective.

Finally, the size of the program is presented on the table below.

Program size (bytes)			
	ROM		RAM
	text	data	bss
Publisher	54605	332	7026
Receiver	54445	332	6924

Table 5.16: Option 4 – Program size

A more detailed view of the power consumption is shown on the following graphs:

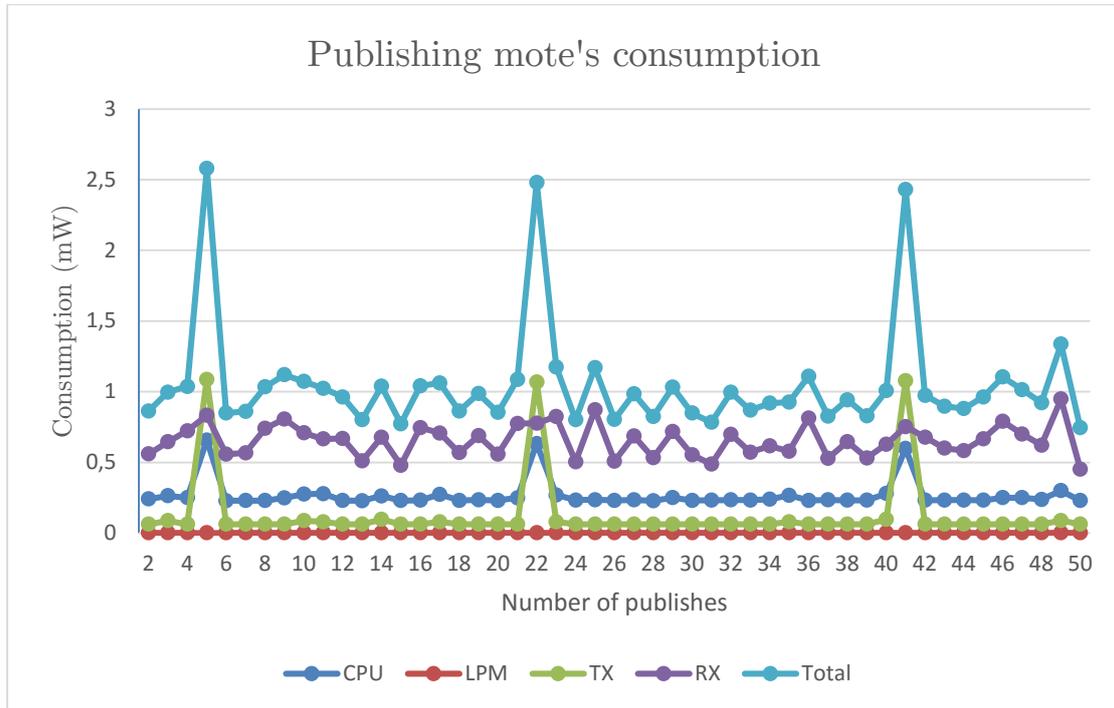


Figure 5.11: Option 4 – Publishing mote’s consumption graph

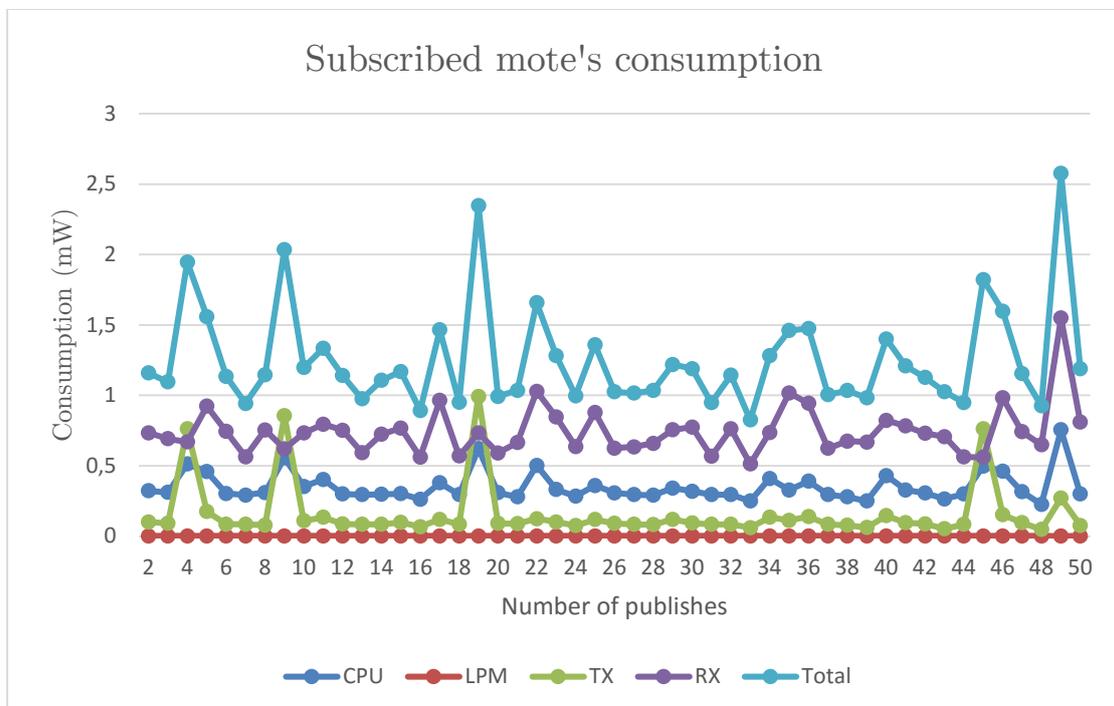


Figure 5.12: Option 4 – Subscribed mote’s consumption graph

5.7. Comparison

This is the most interesting part of the evaluation process, because a comparison between the different options will be presented in an easy to understand way, with the use of bar charts. To begin with, a comparison of all the encryption options when using the same size of message payload (48 bytes) will be done. The only exception in this case is the single block AES that is limited to 16 bytes only. Consequently, a comparison between AES-CBC, AES-OCB and LLSec with different sizes of payload will be presented. Finally, additional comparison charts could be found at Annex A (p.68) of this thesis.

The following bar graphs display the average power consumption for every option as described above in every one of the two MQTT clients used. As a reminder, all the security mechanisms, with the only exception of single block AES, were used in order to encrypt 48 bytes of message payload in this case.

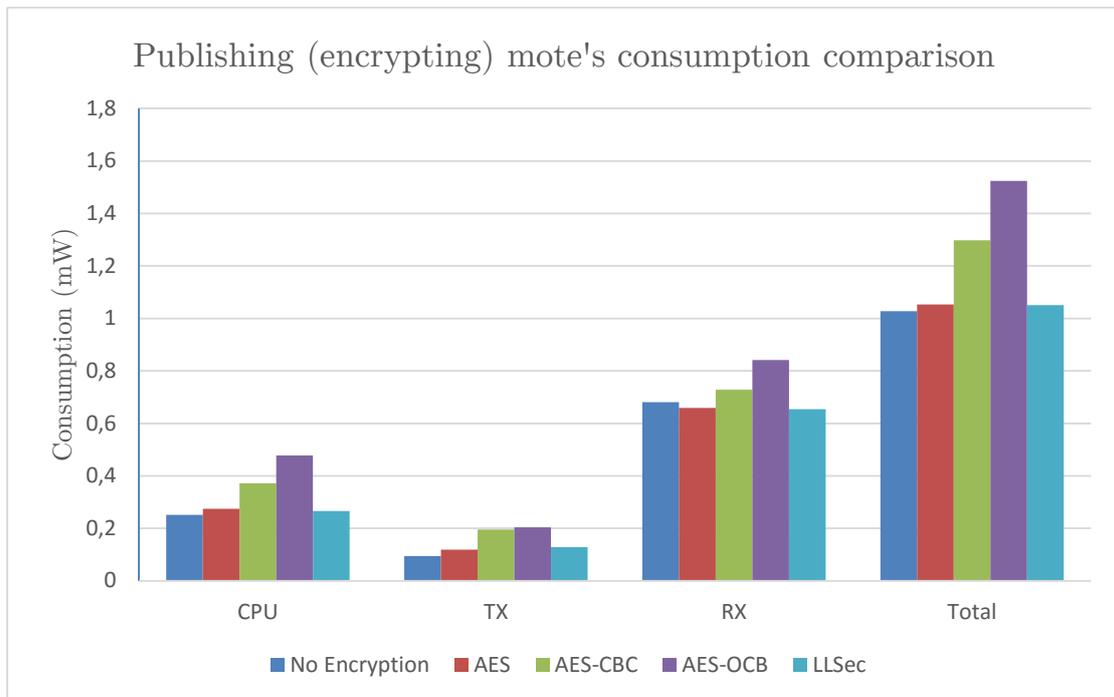


Figure 5.13: Publishing mote's consumption comparison graph

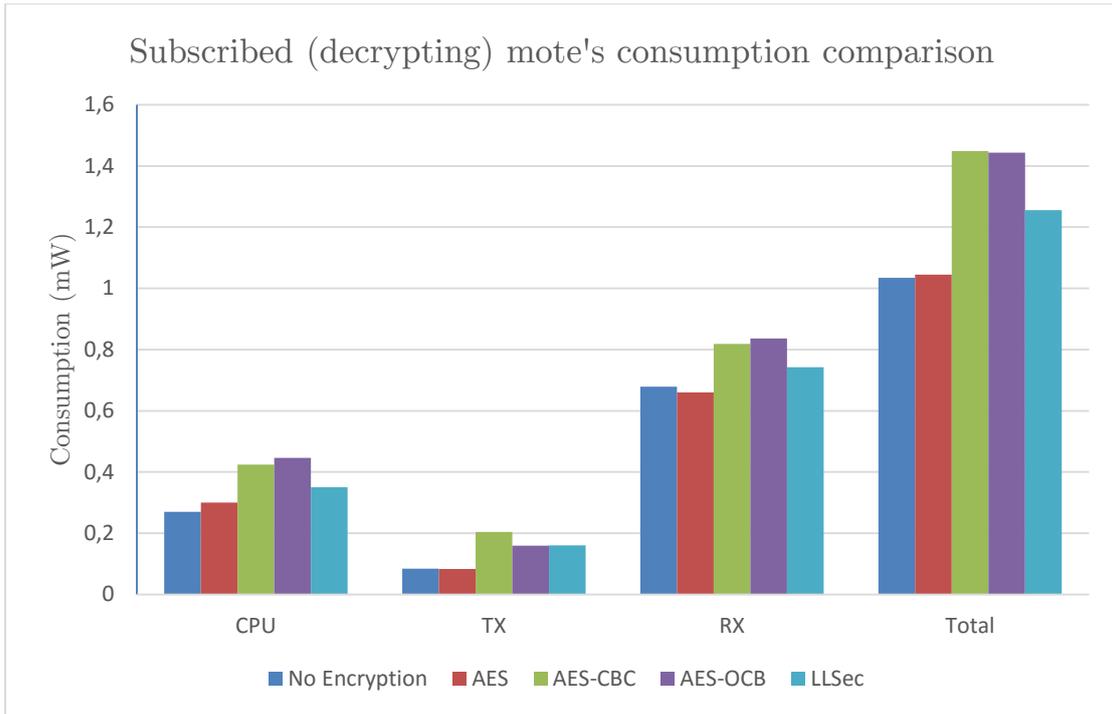


Figure 5.14: Subscribed mote's consumption comparison graph

The average message latency on the same case is also presented on the graph below:

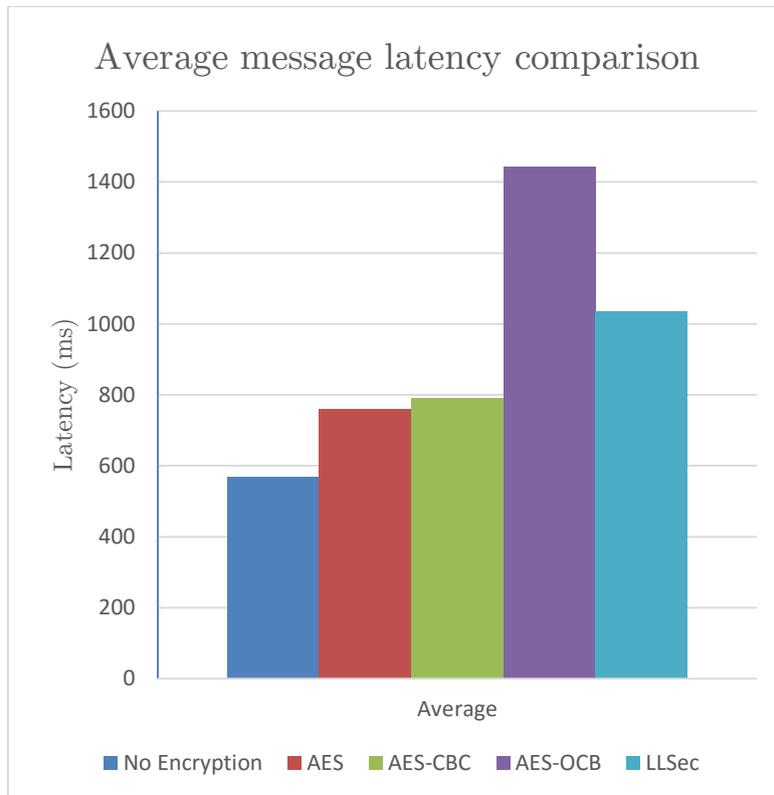


Figure 5.15: Average message latency comparison graph

It is easy to see that AES-OCB is the most resource intensive option when it comes to encryption, as was of course expected due to the complexity of the security mechanism, with the AES-CBC to follow up second. On the decryption side, AES-CBC is pretty close to AES-OCB with the latter requiring smaller CPU time. One thing that is quite unexpected is that LLSec does not have such a large performance impact. It must be also said that the CC2420 radio IC used in Zolertia Z1 has hardware accelerated AES encryption capabilities and the hardware AES driver is selected by default from Contiki if supported by the hardware platform. This is the reason why no huge differences in the CPU usage and more specifically in message latency was observed when using single block AES in comparison with AES-CBC and LLSec as both make use of that capabilities.

When it comes to latency, AES-OCB has unexceptionally the largest average latency due to the previously mentioned complexity of its mechanism and LLSec comes second on the largest average latency rank. This is due to the also complex mechanism used in LLSec; the AES-CCM* which also is an authenticated encryption mechanism. In general, AES-CCM is more resource intensive compared to AES-OCB, but there are two parameters that make LLSec, and therefore AES-CCM*, a lighter option; i) security is established on the Link Layer which is, generally, faster in comparison with security established on the Application Layer, and ii) AES-CCM* is a more lightweight mechanism because it is specially designed for use in constrained devices. On the other hand, there is one parameter that makes LLSec having such a greater average message latency in comparison to AES-CBC; that is due to the fact that it provides node-to-node encryption, thus performing decryption and encryption on every hop (e.g. first in the publishing mote, then in the slip-radio mote, and finally, again, in the receiving mote) of its route, extending with that way the total message latency.

The next two graphs show a visual comparison of the program sizes of each option in both ROM and RAM of the hardware platform:

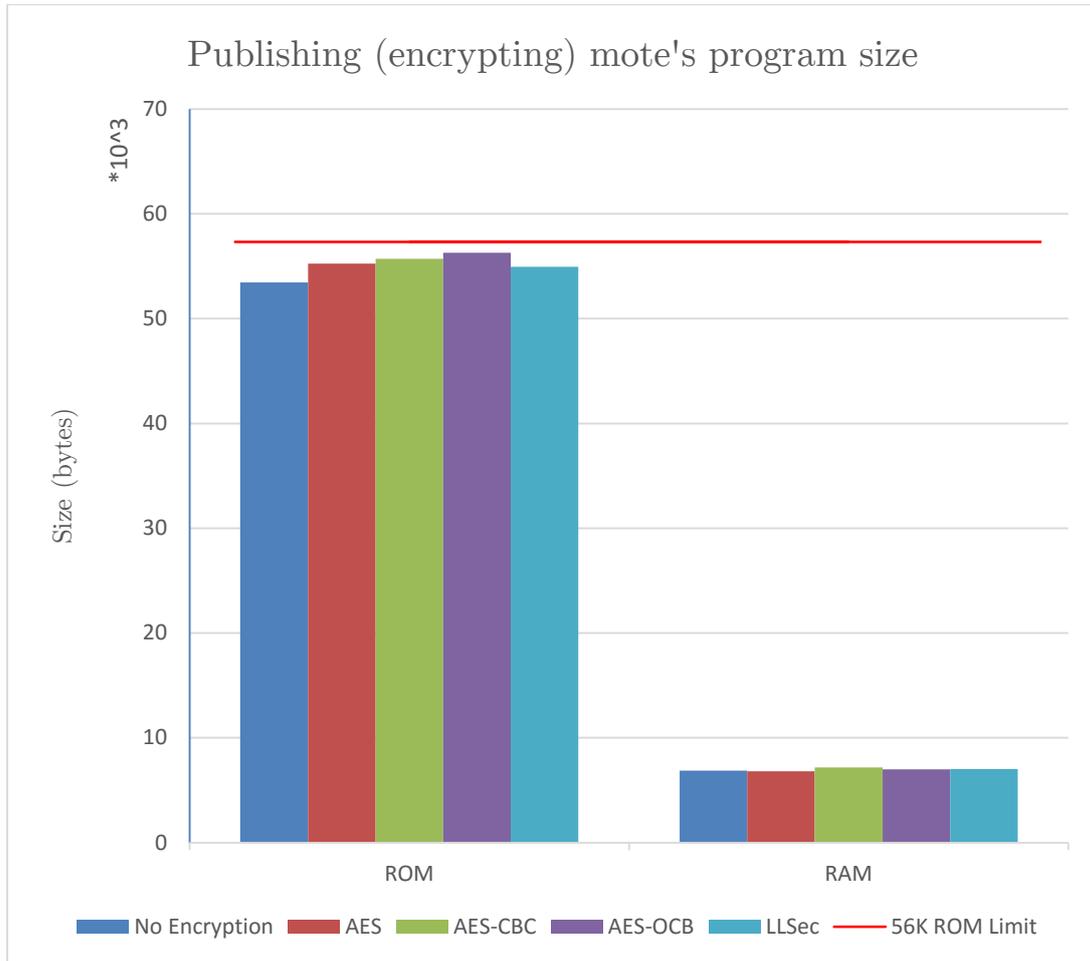


Figure 5.16: Publishing mote's program size comparison graph

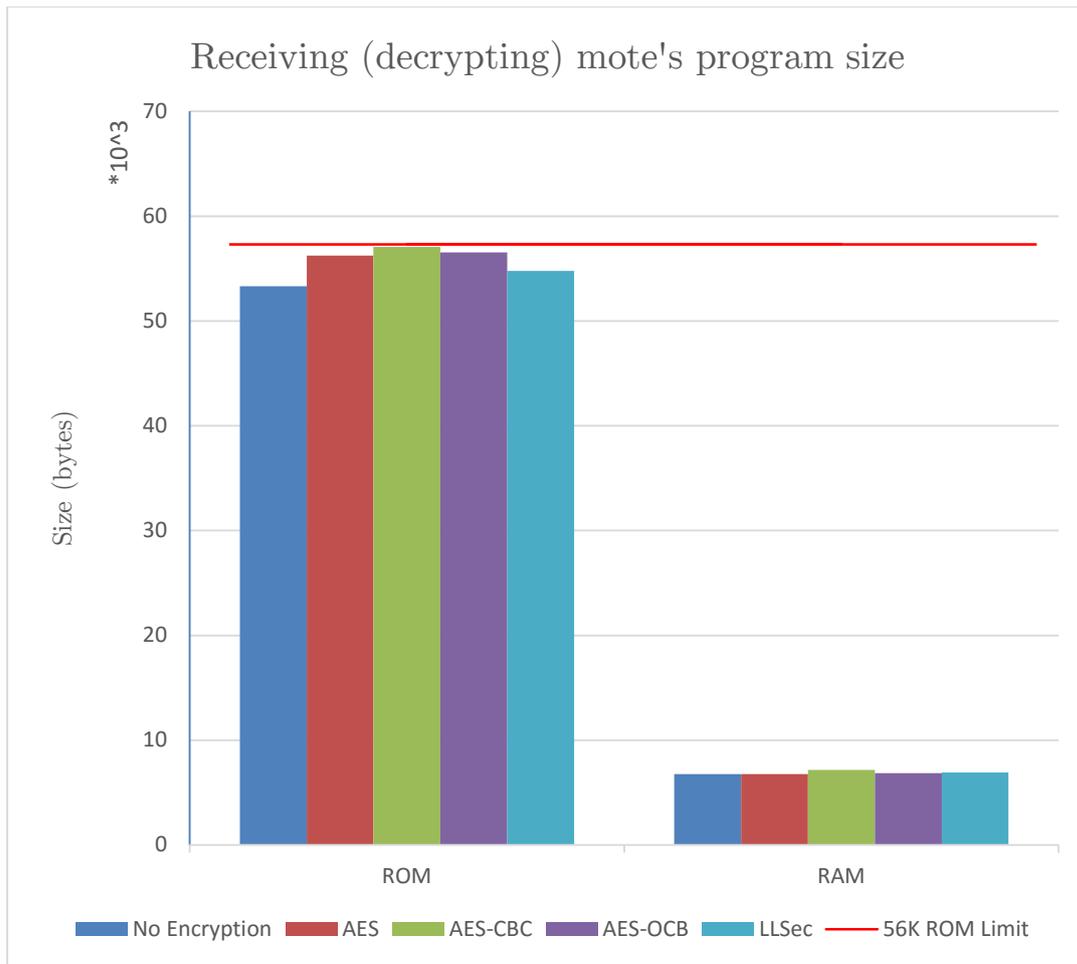


Figure 5.17: Receiving mote's program size comparison graph

In the above two graphs the 56 Kilobytes limit of the ROM when using the stable version 4.7.0 of the msp430-gcc compiler is also shown as a reference. It must be emphasized that the AES-CBC algorithm almost filled up all the available space on both ROM and RAM in the receiving (decrypting) mote.

Finally, another comparison that has some interest is the one between the three main different security mechanisms when using different message payload sizes. The results of the above comparison are shown on the next graph:

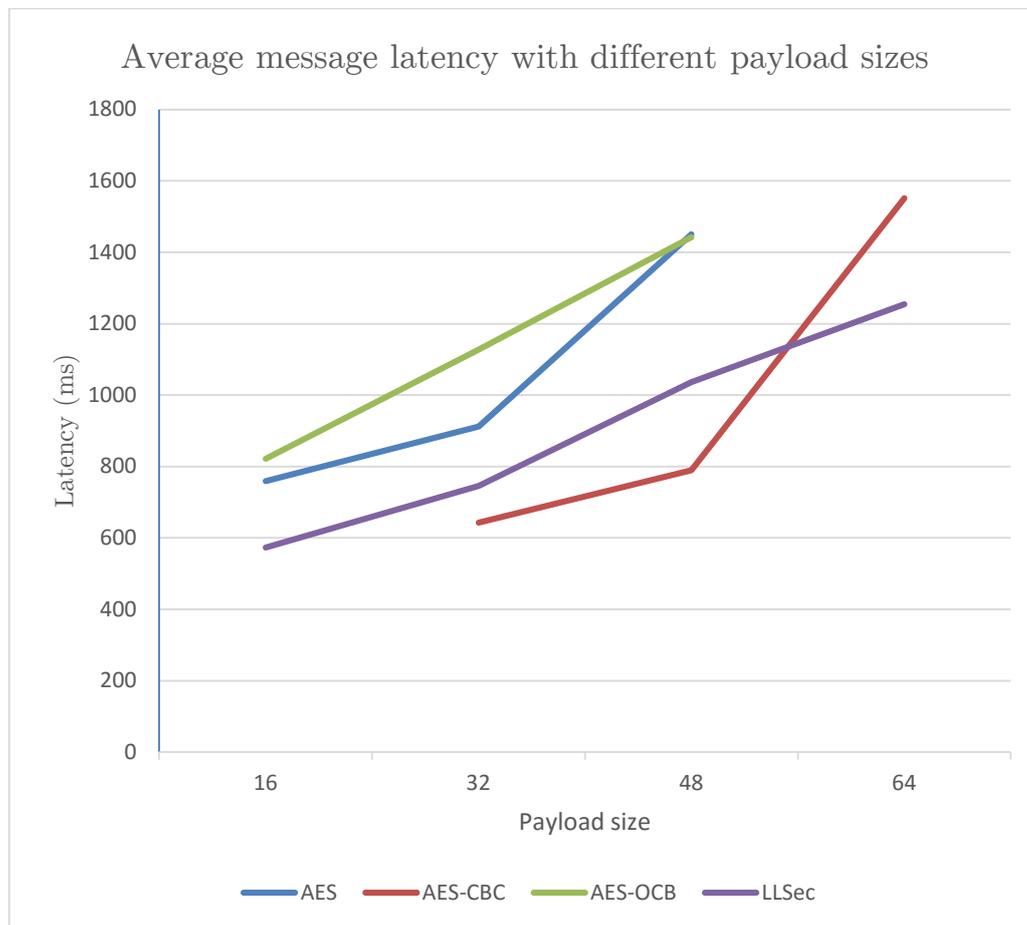


Figure 5.18: Average message latency vs payload size comparison graph

As it is expected, greater message payload sizes entail greater message latency times due to both larger encryption/decryption time but also larger packet trip time. However, in all cases AES-OCB had the largest latency, except when using AES-CBC with the enormous payload size of 64 bytes, while the AES-CBC had undoubtedly the best latency-to-payload size ratio. One more observation is that, when using the single block AES in manual ECB mode (with consecutive encryptions/decryption), it consumes the most resources on every case (i.e. 32 bytes and 48 bytes of payload, as seen on the above figure) and has the greater message latency (due to the continuous encryptions/decryptions) making it the less suitable option. For more comparison charts refer to Annex A of this thesis.

As a last note, it should be stated that the average time needed by a real environment mote, for establishing its first connection to the MQTT broker was measured at about 12 seconds while the average reconnection time was measured at about 20 seconds.

6. Conclusions and Open Issues

6.1. Conclusions

In this work, four different secure and lightweight MQTT client implementations for the Contiki OS were developed and evaluated on a real IoT hardware platform. These implementations were based on security mechanisms on two separate layers of the OSI Model; the Application Layer, and the Link Layer. Other tools, such as, the 6lbr 6LoWPAN Border Router and the Mosquitto MQTT broker were also used for deploying a full IoT system that uses MQTT for communication. Furthermore, a feature comparison of the various IoT messaging protocols had been carried out during the literature study phase of this thesis.

The three (options 2 to 4) of the four different secure and lightweight implementations deployed are ready for use on real IoT applications and could be used as a basis for future implementation or studies. Moreover, the feature comparison of the communication protocols could be used as a compass for selecting the right one according to the application's needs.

Finally, when it comes to the selection for the better security implementation for MQTT, both AES-CBC and LLSec are the best two options with each and every having some advantages over the other. The first one has the advantage of the best payload-to-latency ratio, according to the evaluation done, and provides end-to-end encryption while the second one offers node-to-node encryption, is the less resource consuming option and the easier one to use.

6.2. Problems encountered

During the development phase some problems were encountered, especially with the proper configuration of 6lbr and also with some bugs on the msp430-gcc compiler that prevented us from using the full 92 Kilobytes of flash memory available on the Zolertia Z1. While the latter was not a major issue for us, it must be taken care of for future expansion of the above implementations because some of them have almost reached the ROM usage limit of 56 Kilobytes. However, having solved the aforementioned problem, we should be able to evaluate the performance of some mechanisms with even larger message payload sizes (for example AES-OCB and AES-ECB with 64bytes payload that were not performed due to limited ROM). The problem with 6lbr on the other hand has been resolved by deploying it on a Raspberry Pi instead of running it on a VM. However, 6lbr had some issues with LLSec, that are, hopefully, resolved in the latest version 1.4.0 which was released some days before. Finally, there are of course a lot of other simple bugs that prevented us of using the latest commit

of Contiki and must be fixed. This is of course expected due to the open source and community driven nature of the Contiki OS where everyone can contribute to it.

As an additional note on this chapter, the lessons learned from this work will be mentioned. Firstly, it should be noted that a slightly large amount of time was used for researching the available MQTT client implementations for Contiki as well as the available security mechanisms in Contiki. This, however, could be bypassed if Contiki had a continuously updated and detailed list of the built in support of protocols and applications.

The even greater problem, that consumed a great amount of time, was the process of reducing the source code of the MQTT client in order to make enough free space on ROM and RAM for the security mechanism that will be later added. The Contiki's MQTT client source code was flawlessly written, but was probably designed for use as is, without the addition of extra features, because, in its original version, it consumed enough redundant space in ROM in the sake of easier configuration and easier reading. A simple workaround on this issue, was of course, the use of a more powerful hardware platform that has more flash memory and RAM, but that was not possible in our case. Another not so fast solution, could be the development of a MQTT client from scratch based only on the MQTT library of Contiki. This was, though not done due to the limited time available for this thesis and because the focus was on the many different secure and lightweight implementations of MQTT in Contiki and not in developing only one, more lightweight, but not secure MQTT client implementation.

Finally, a significant amount of time was also spent on trying to deploy this MQTT system in the real world testing environment, namely in real motes, and getting them to connect to the MQTT broker running on a LAN connected machine. For this to succeed, a lot of different network configuration (including different configurations for the IPv6 stack and the devices addressing) were tested on both the 6lbr as well as the LAN router used. Finally, but after some time, the better for our case was found. All this time could have been avoided if 6lbr had a complete documentation describing all the modes of operation it supports and how it must be configured in different situations. The current documentation of 6lbr is unfortunately, not up to date and has many missing entries.

6.3. Open Issues

In the future, we intend to develop more secure MQTT implementations for use on the IoT by exploiting even more layers of the OSI Model such as the Network Layer, with the use of the compressed version of IPSec designed for constrained devices, or the Transport Layer, with the use of, the most popular, TLS. Furthermore, a more comprehensive comparison of the IoT protocols with the performance assessment of them could also be done.

On a topic not strictly related to this thesis, a great area of research, where much work still needs to be done, is the interoperability in IoT. As already mentioned, IoT contains very different communication, and not only, protocols. As a result, a method for enabling IoT devices using different communication protocols to communicate with each other, either by using a “smart” bridge or by using protocol conversion/translation on the application layer, would be really groundbreaking.

Bibliography

- [1] S. C. Mukhopadhyay, “Internet of things: challenges and opportunities”, vol. 9.; 9, pp. 1{7. Springer, 2014
- [2] Cisco IBSG, 2011.
- [3] Dave Evans, “The Internet of Things: How the Next Evolution of the Internet Is Changing Everything”, *Cisco*, April 2011. [Online]. Accessed: 6 May 2016. Available: http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.
- [4] K. Rantos, A. Papanikolaou, C. Manifavas and I. Papaefstathiou, “IPv6 Security for Low Power and Lossy Networks”, *IEEE IFIP Wireless Days (WD)*, 2013.
- [5] S. Raza, T. Chung, S. Duquennoy, D. Yazar, T. Voigt and U. Roedig, “Securing Internet of Things with Lightweight IPsec”, *SICS Technical Report*, August 2010.
- [6] A. Banks, R. Gupta, “OASIS Message Queuing Telemetry Transport (MQTT), version 3.1.1”, *OASIS*, 2015. [Online]. Accessed: December 2015. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.pdf>.
- [7] Contiki: The Open Source OS for the Internet of Things. [Online]. Available: <http://www.contiki-os.org>.
- [8] Zolertia Z1 platform, *Zolertia*. [Online]. Available: <http://zolertia.io/z1>.
- [9] 6lbr: A deployment-ready 6LoWPAN Border Router solution based on Contiki. [Online]. Available: <http://cetic.github.io/6lbr>.
- [10] Kevin Ashton, “That 'Internet of Things' Thing”, *RFID Journal*, 22 June 2009.
- [11] “Radio-Frequency Identification”, *Wikipedia, the Free Encyclopedia*. [Online]. Accessed: May 2016. Available: http://en.wikipedia.org/wiki/Radio-frequency_identification.
- [12] K. Rose, S. Eldridge and L. Chapin, “The Internet of Things: An Overview Understanding the Issues and Challenges of a More Connected World”, *The Internet Society (ISOC)*, October 2015.
- [13] “RFC 2460: Internet Protocol, Version 6 (IPv6) Specification”, *The Internet Engineering Task Force (IETF)*. [Online]. Available: <https://tools.ietf.org/html/rfc2460>.

- [14] G. E. Moore, "Moore's Law", *Wikipedia, the Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Moore's_law.
- [15] IEEE Standards Association [Online]. Available: <http://standards.ieee.org/innovate/iot>.
- [16] AllSeen Alliance [Online]. Available: <https://allseenalliance.org>.
- [17] OASIS consortium [Online]. Available: https://www.oasis-open.org/committees/tc_cat.php?cat=iot.
- [18] Industrial Internet Consortium [Online]. Available: <http://www.iiconsortium.org>.
- [19] K. Rose, S. Eldridge, L. Chapin "The Internet of Things: An Overview - Understanding the Issues and Challenges of a More Connected World", *The Internet Society (ISOC)*, October 2015. [Online]. Available: <http://www.internetsociety.org/sites/default/files/ISOC-IoT-Overview-20151022.pdf>
- [20] White Paper, "Internet of Things: Wireless Sensor Networks", *International Electrotechnical Commission (IEC)*. [Online]. Available: <http://www.iec.ch/whitepaper/pdf/iecWP-internetofthings-LR-en.pdf>.
- [21] "IEEE 802.15: Wireless Personal Area Networks (PANs)", *IEEE Standards Association*. [Online]. Available: <http://standards.ieee.org/about/get/802/802.15.html>.
- [22] "Carrier sense multiple access with collision avoidance", *Wikipedia, the Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_avoidance.
- [23] "IPv6 over Low power WPAN (6lowpan)", *The Internet Engineering Task Force (IETF)*. [Online]. Available: <https://datatracker.ietf.org/wg/6lowpan/documents>.
- [24] Mulligan, Geoff, "The 6LoWPAN architecture", *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors, ACM*, 2007.
- [25] Z. Shelby and C. Bormann, "6LoWPAN: The wireless embedded Internet - Part 1: Why 6LoWPAN?", *EE Times*, 23 May 2011.
- [26] A. G. Blank, "TCP/IP Jumpstart: Internet protocol basics". *John Wiley & Sons*, 2006.
- [27] OASIS. [Online]. Available: <https://www.oasis-open.org>.

- [28] Y. Xiao, H.H. Chen, B. Sun, R. Wang and S. Sethi, "MAC security and security overhead analysis in the IEEE 802.15.4 Wireless Sensor Networks", *EURASIP Journal on Wireless Communications and Networking*, 2006.
- [29] J. Reardon and I. Goldberg, "Improving Tor using a TCP-over-DTLS Tunnel", *USENIX Security Symposium*, 2009. [Online]. Available: https://www.usenix.org/legacy/event/sec09/tech/full_papers/reardon.pdf.
- [30] ACL definition, *Wikipedia, the Free Encyclopedia*. [Online]. Available: http://en.wikipedia.org/wiki/Access_control_list.
- [31] Figure is courtesy of HiveMQ. [Online]. Available: <http://www.hivemq.com/blog/mqtt-security-fundamentals-payload-encryption>.
- [32] Blog post, "MQTT Security Fundamentals: OAuth 2.0 & MQTT", *HiveMQ blog*. [Online]. Available: <http://www.hivemq.com/blog/mqtt-security-fundamentals-oauth-2-0-mqtt>.
- [33] J. Bregell, "Hardware and software platform for Internet of Things", *Master of Science Thesis in Embedded Electronic System Design*, 2015.
- [34] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems", pp. 29-42, *ACM*, 2006.
- [35] Raspberry Pi boards, *Raspberry Pi Foundation*. [Online]. Available: <https://www.raspberrypi.org/products>.
- [36] Mosquitto Open Source MQTT v3.1/v3.1.1 Broker. [Online]. Available: <http://mosquitto.org>.
- [37] L. Zhang, "Building Facebook Messenger", *Faceboo*, 12 August 2011. [Online]. Available: <https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920>
- [38] Zolertia, "Zolertia Z1 Datasheet". [Online]. Available: http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf.
- [39] Instant Contiki. [Online]. Available: <http://www.contiki-os.org/start.html>.
- [40] Contiki, "The ContikiMAC Radio Duty Cycling Mechanism". [Online]. Available: https://github.com/contiki-os/contiki/wiki/Radio-duty-cycling#The_ContikiMAC_Radio_Duty_Cycling_Mechanism.

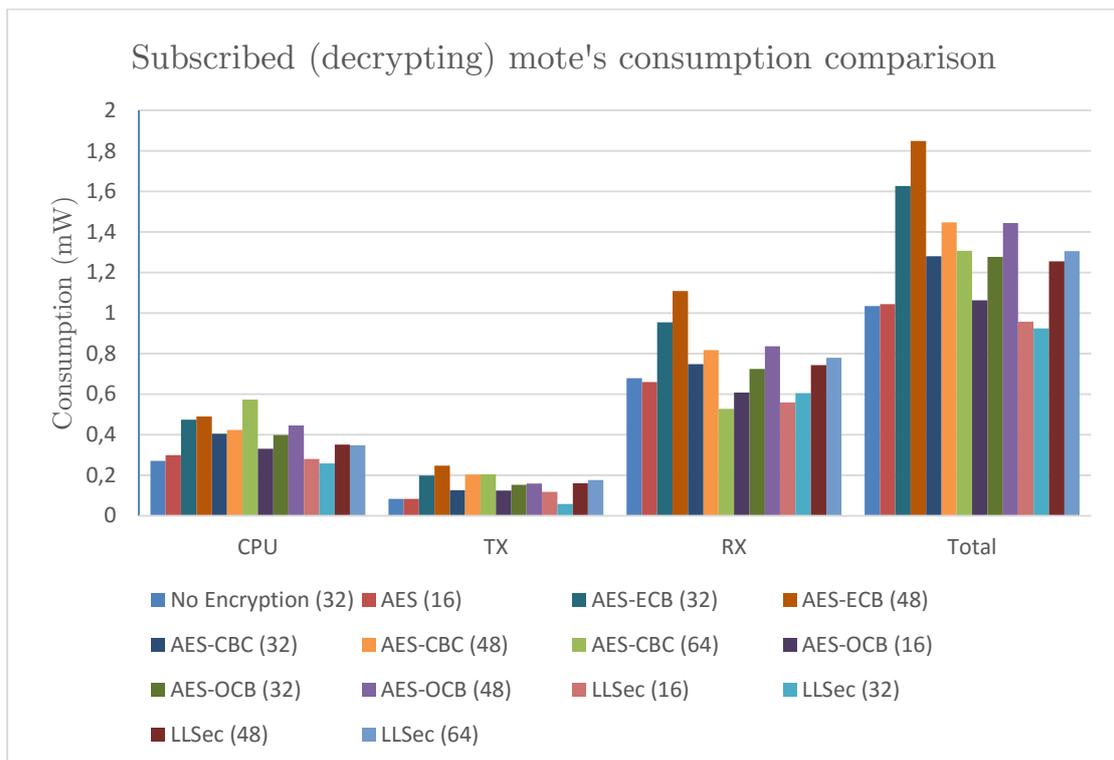
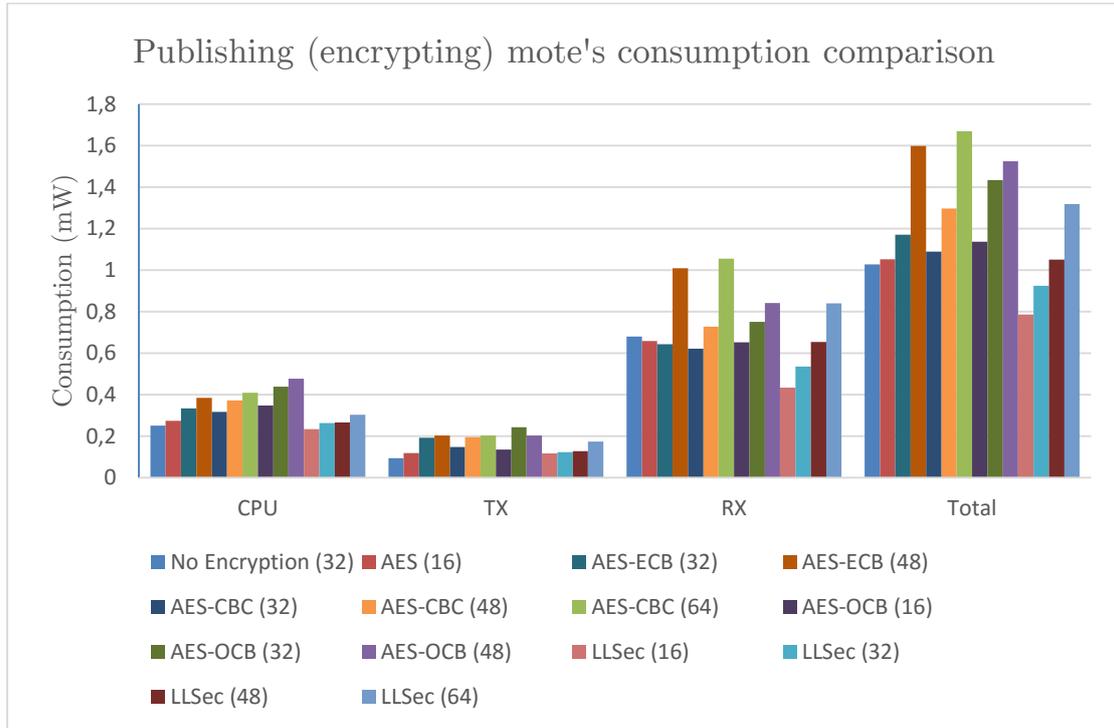
- [41] “ISO/IEC 20922 Information technology -- Message Queuing Telemetry Transport (MQTT) v3.1.1”, *ISO*, 2016. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=69466
- [42] Andy Stanford-Clark, “Why is the keep alive needed?”, *Google Groups*. [Online]. Available: <https://groups.google.com/forum/#!msg/mqtt/zRqd8JbY4oM/XrMwLQ5TU0EJ>.
- [43] “Block cipher mode of operation”, *Wikipedia, the Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation.
- [44] “RFC 3610: Counter with CBC-MAC (CCM)”, *The Internet Engineering Task Force (IETF)*. [Online]. Available: <https://tools.ietf.org/html/rfc3610>.
- [45] “OCB: Free licenses”, *OCB webpage*. [Online]. Available: <http://web.cs.ucdavis.edu/~rogaway/ocb/license.htm>.
- [46] “Kerberos: The Network Authentication Protocol”, *Kerberos website*. [Online]. Available: <http://web.mit.edu/kerberos>.
- [47] T. Nixon and A. Regnier, “Devices Profile for Web Services (DPWS), version 1.1”, *OASIS*, July 2009. [Online]. Available: <http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01>.
- [48] “EU Research Project SOCRADES 2006-2009”. [Online]. Available: <http://www.socrades.net>.
- [49] “RFC 7622: Extensible Messaging and Presence Protocol (XMPP): Address Format”, *The Internet Engineering Task Force (IETF)*. [Online]. Available: <https://tools.ietf.org/html/rfc7622>.
- [50] “RFC 7252: The Constrained Application Protocol (CoAP)”, *The Internet Engineering Task Force (IETF)*. [Online]. Available: <https://tools.ietf.org/html/rfc7252>.
- [51] U. Kretschmar, “AES128 – A C Implementation for Encryption and Decryption (Rev. A)”, *Texas Instruments*, 2009. [Online]. Available: <http://www.ti.com/mcu/docs/litabsmultiplefilelist.tsp?sectionId=96&tabId=1502&literatureNumber=slaa397a&docCategoryId=1&familyId=914>.
- [52] P. Tsigas and L. Casado, "ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System", *14th*

- Nordic Conference on Secure IT Systems*, 2009. [Online]. Available:
<http://www.cse.chalmers.se/research/group/dcs/masters/contikisec>.
- [53] I. Skerrett, “IoT Developer Survey 2016”, *Eclipse IoT Working Group, IEEE IoT and Agile IoT*, April 2016. [Online]. Available:
<http://www.slideshare.net/IanSkerrett/iot-developer-survey-2016>.
- [54] Texas Instruments, “MSP430F261x and MSP430F241x Mixed Signal Microcontroller Datasheet”, November 2012. [Online]. Available:
<http://www.ti.com/lit/ds/symlink/msp430f2417.pdf>.
- [55] V. Jutvik, “IPsec and IKEv2 for the Contiki Operating System”, *Uppsala University, Master Thesis*, June 2014. [Online]. Available:
<http://uu.diva-portal.org/smash/get/diva2:736994/FULLTEXT01.pdf>

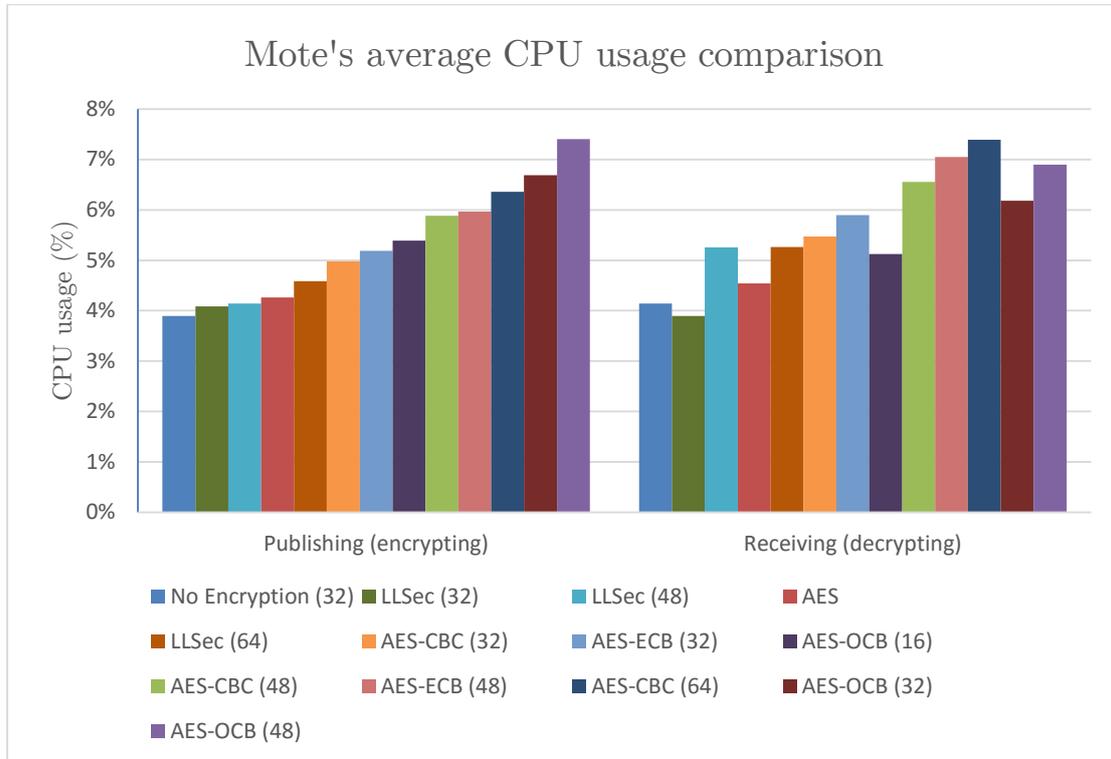
Annex A

Extended comparison of all the options

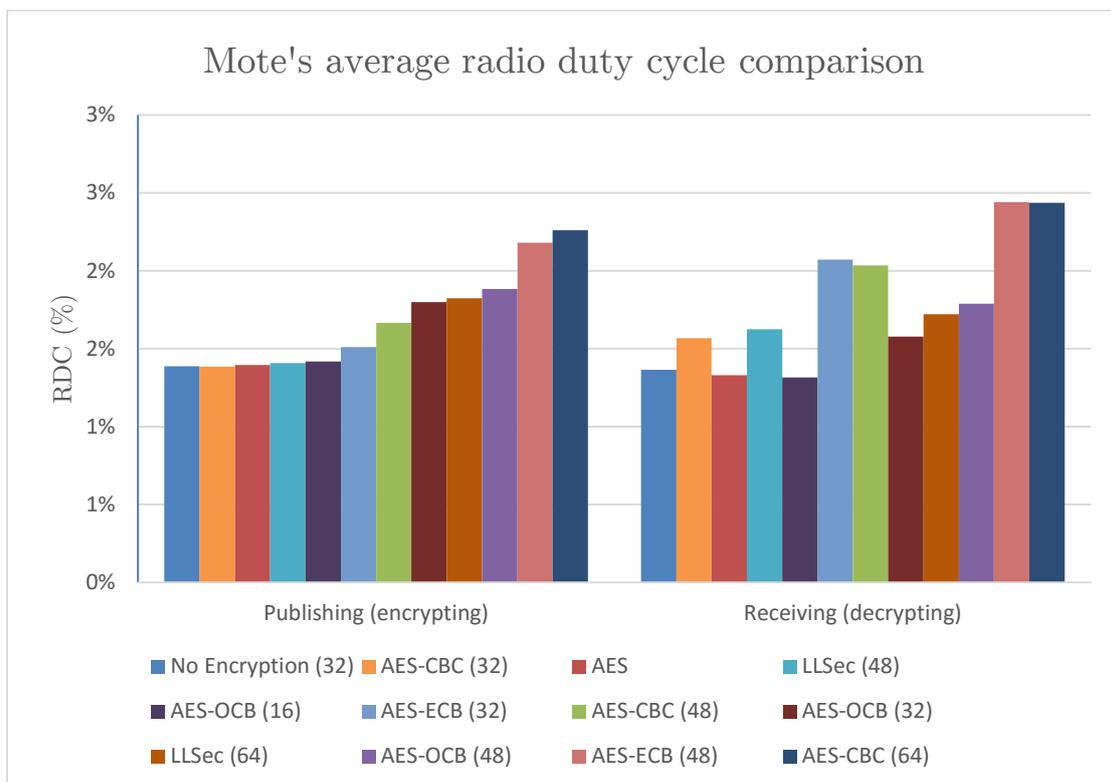
On the bellow two graphs an extended comparison of all the options with various payload sizes is done



A comparison of the CPU usage has also been made and the results are displayed on the following graph:



Finally, the following table displays the average radio duty cycle comparison:



Option 1b – single block AES used in manual ECB mode

In the following tables and graphs, the performance evaluation of the single block AES mechanism when used in manual ECB mode (consecutive encryptions/decryptions) will be presented. The following data are for the encryption of 48 bytes of message payload.

The average power consumptions and the radio duty cycle are presented on the following tables:

Average power consumption (mW)					
	CPU	LPM	TX	RX	Total
Publisher (encryption)	0,384270	0,001455	0,203498	1,009919	1,599143
Receiver (decryption)	0,490507	0,001443	0,247990	1,108410	1,848350

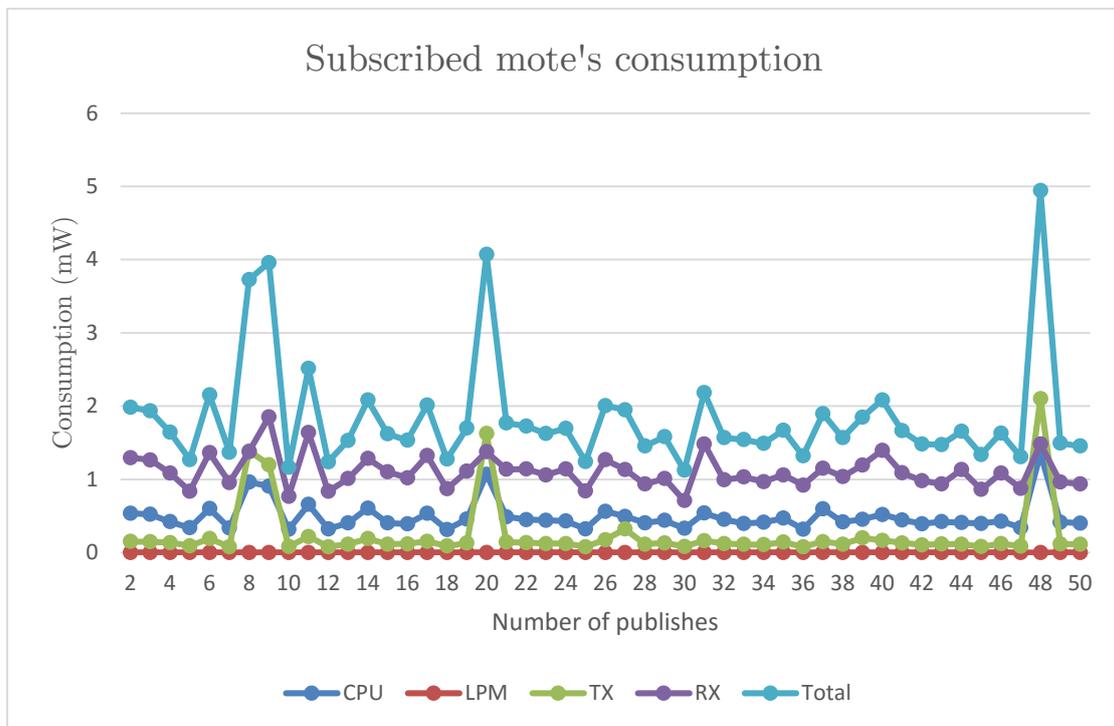
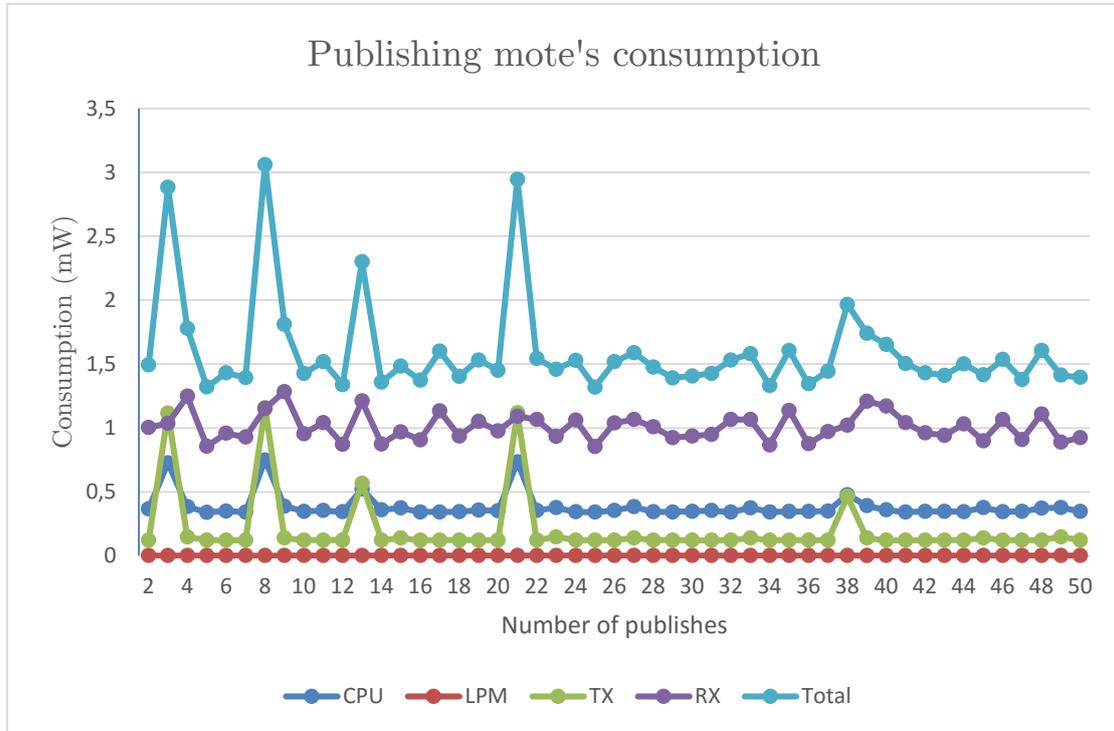
Average radio duty cycle (%)			
	TX	RX	Total
Publisher	0,39%	1,79%	2,18%
Receiver	0,48%	1,97%	2,44%

The average latency of the messages when using this mechanism in this mode was extremely larger than before due to the time needed for the consecutive encryptions and decryptions of the messages and was **1450,6 ms**. Moreover, the average time needed for the encryption of the message was calculated on the virtual evaluation environment to be about 32 ms. It must be noted that this specific implementation (with 48 bytes of message payload) is the worst one when it comes in performance and power consumption comparison.

Finally, the size of the program on the hardware platform is presented on the table below.

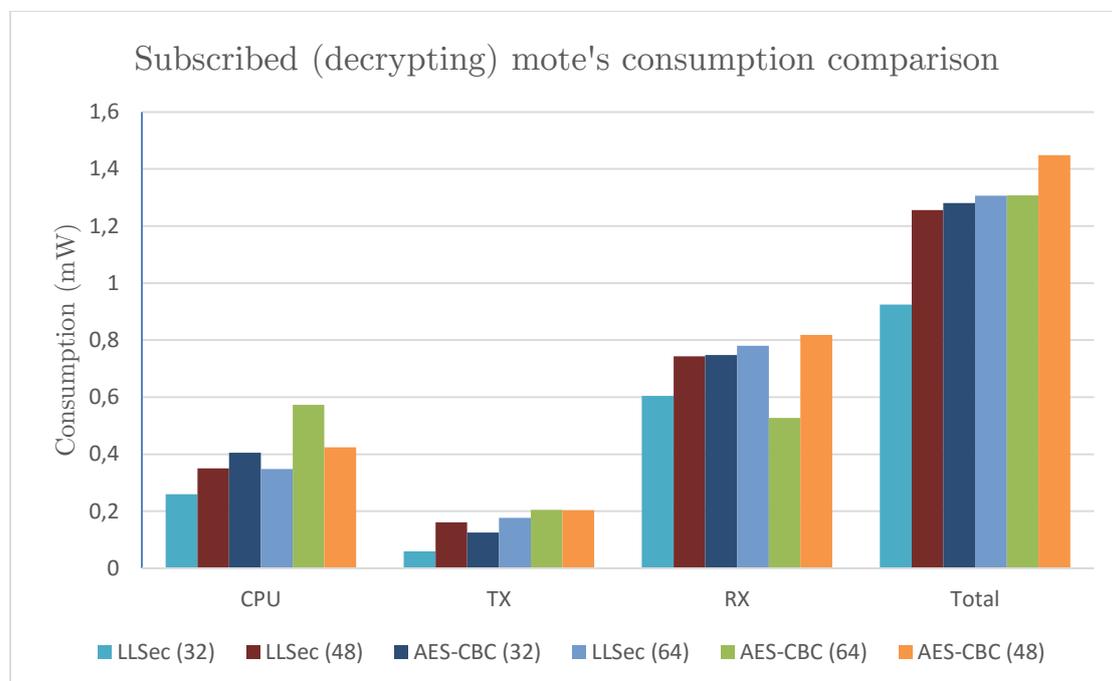
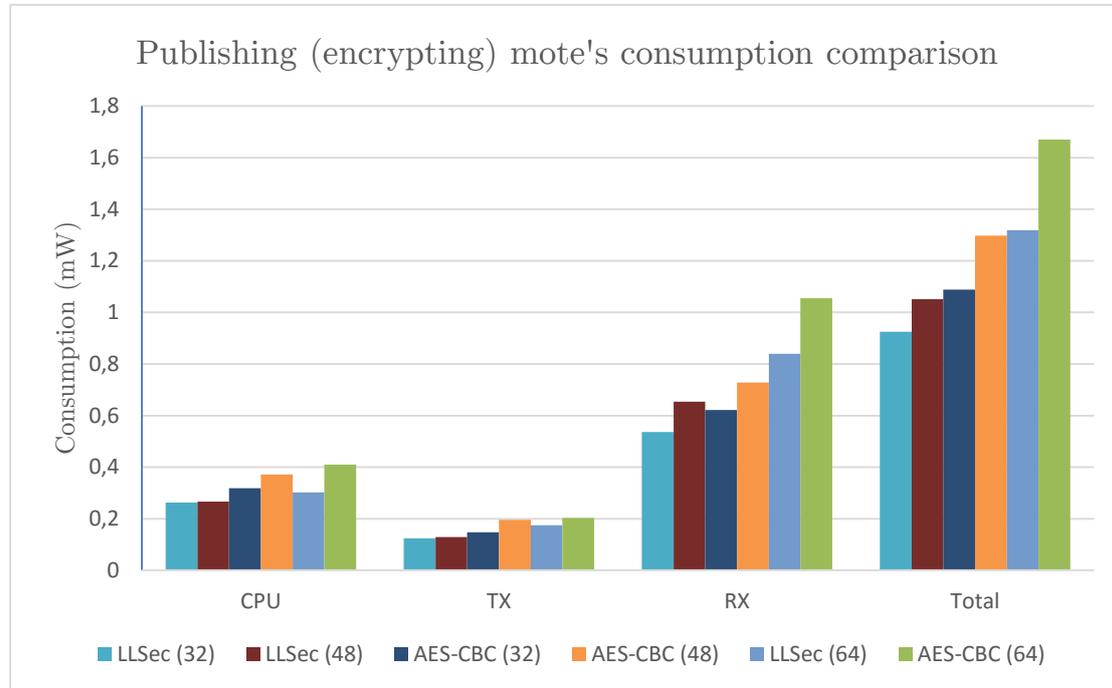
Program size (bytes)			
	ROM		RAM
	text	data	bss
Publisher	55377	322	6862
Receiver	56265	322	6780

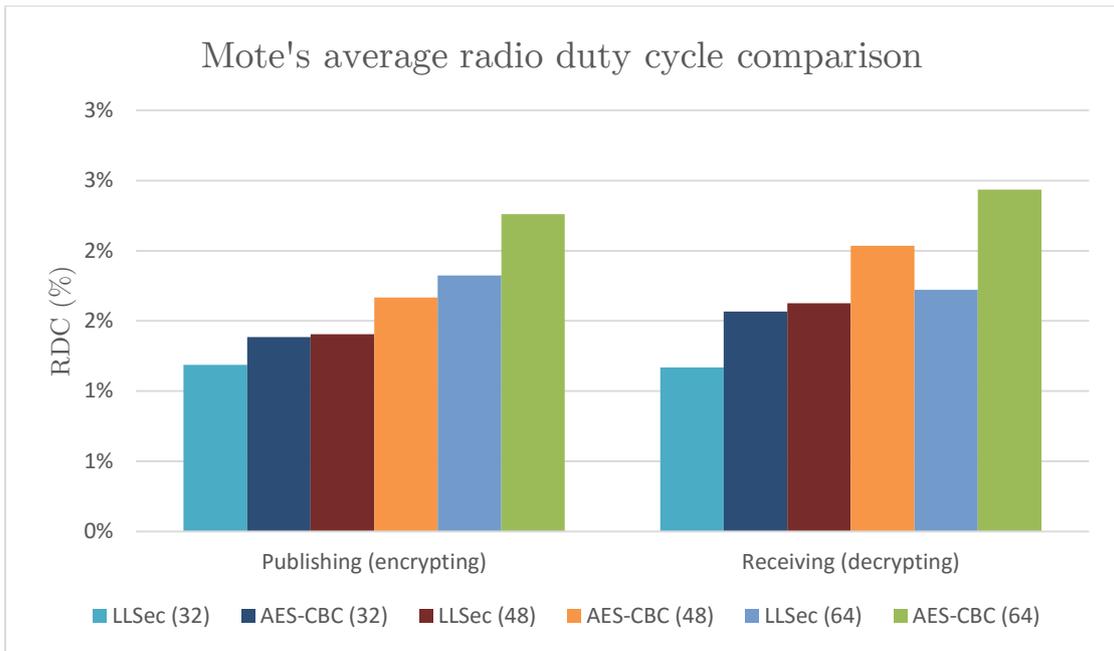
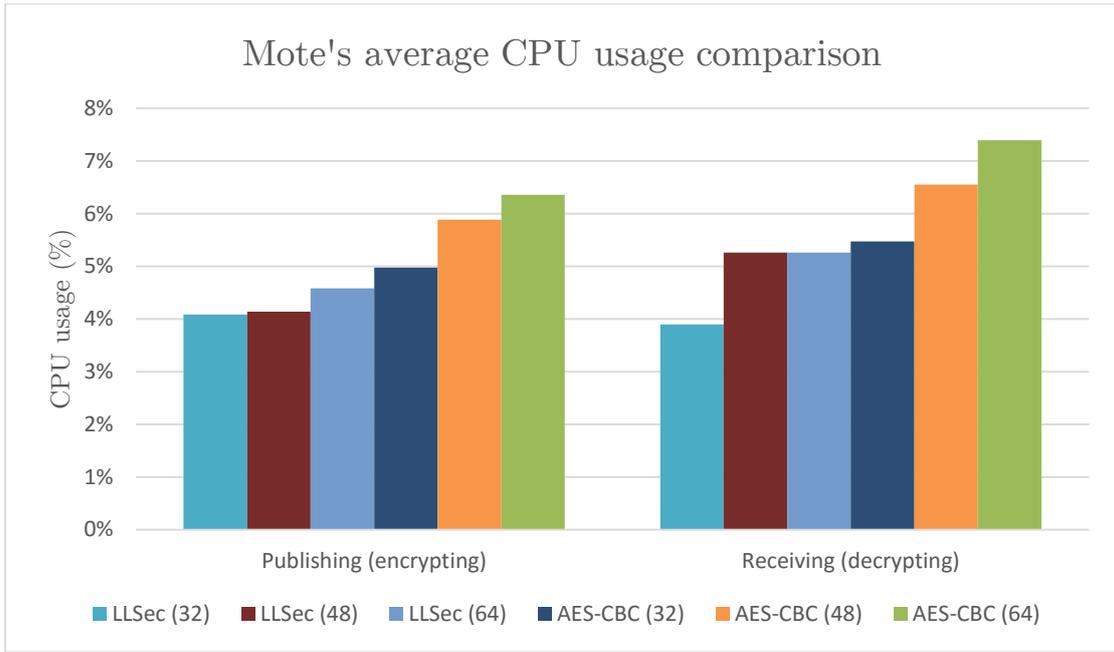
A more detailed view of the power consumption is shown on the following graphs:



Comparison of the two suggested mechanisms

Finally, a focused comparison of the two suggested mechanisms on this thesis, namely the AES-CBC and the LLSec, will be presented at this point.





Annex B

The source code used for the implementations is quoted in this final annex. As a reminder, the source code is based on the MQTT client example of the Contiki's source code. The snippets that are highlighted with yellow color contain the source code for the security implementations.

➤ Simple MQTT client (option 0):

```
/*-----  
*           Author: Sotiris Katsikeas           *  
*           Date: April 2016                   *  
*           Simple MQTT client version with status led - REDUCED           *  
*-----*/  
#include "contiki-conf.h"  
#include "mqtt.h"  
#include "net/ipv6/sicslowpan.h"  
#include "net/llsec/noncoresec/noncoresec.c" // Only for llsec  
#include "dev/leds.h"  
#include <string.h>  
/*-----*/  
/*  
* Publish to a MQTT broker (e.g. mosquitto) running on the host provided by  
MQTT_DEMO_BROKER_IP_ADDR  
*/  
#define DEFAULT_ORG_ID "mqtt-demo"  
/*-----*/  
/*  
* A timeout used when waiting for something to happen (e.g. to connect or to  
* disconnect)  
*/  
#define STATE_MACHINE_PERIODIC (CLOCK_SECOND >> 1)  
/*-----*/  
/* Provide visible feedback via LEDS during various states */  
/* When connecting to broker */  
#define CONNECTING_LED_DURATION (CLOCK_SECOND >> 2)  
  
/* Each time we try to publish */  
#define PUBLISH_LED_ON_DURATION (CLOCK_SECOND)  
/*-----*/  
/* Connections and reconnections */  
#define RETRY_FOREVER 0xFF  
#define RECONNECT_INTERVAL (CLOCK_SECOND * 2)  
  
/*  
* Number of times to try reconnecting to the broker.  
*/
```

```

* Can be a limited number (e.g. 3, 10 etc) or can be set to RETRY_FOREVER
*/
#define RECONNECT_ATTEMPTS      RETRY_FOREVER
#define CONNECTION_STABLE_TIME  (CLOCK_SECOND * 5)
static struct timer connection_life;
static uint8_t connect_attempt;
/*-----*/
/* Various states */
static uint8_t state;
#define STATE_INIT      0
#define STATE_REGISTERED  1
#define STATE_CONNECTING  2
#define STATE_CONNECTED  3
#define STATE_PUBLISHING  4
#define STATE_DISCONNECTED  5
#define STATE_NEWCONFIG  6
#define STATE_CONFIG_ERROR 0xFE
#define STATE_ERROR      0xFF
/*-----*/
#define RSSI_MEASURE_INTERVAL_MAX 86400 /* secs: 1 day */
#define RSSI_MEASURE_INTERVAL_MIN  5 /* secs */
#define PUBLISH_INTERVAL_MAX      86400 /* secs: 1 day */
#define PUBLISH_INTERVAL_MIN      5 /* secs */
/*-----*/
/* A timeout used when waiting to connect to a network */
#define NET_CONNECT_PERIODIC      (CLOCK_SECOND >> 2)
#define NO_NET_LED_DURATION      (NET_CONNECT_PERIODIC >> 1)
/*-----*/
/* Default configuration values */
#define DEFAULT_TYPE_ID           "cc2420"
#define DEFAULT_PUBLISH_TOPIC     "clients/status"
#define DEFAULT_SUBSCRIBE_TOPIC  "test/+"
#define DEFAULT_BROKER_PORT       1883
#define DEFAULT_PUBLISH_INTERVAL (CLOCK_SECOND * 20)
#define DEFAULT_KEEP_ALIVE_TIMER  60
#define DEFAULT_RSSI_MEAS_INTERVAL (CLOCK_SECOND * 30)
/*-----*/
/* Payload length of ICMPv6 echo requests used to measure RSSI with def rt */
#define ECHO_REQ_PAYLOAD_LEN      20
/*-----*/
PROCESS_NAME(mqtt_demo_process);
AUTOSTART_PROCESSES(&mqtt_demo_process);
/*-----*/
/* Maximum TCP segment size for outgoing segments of our socket */
#define MAX_TCP_SEGMENT_SIZE     32
/*-----*/
#define STATUS_LED LEDS_GREEN
#define STATE_LED  LEDS_BLUE
/*-----*/
/*

```

```

* Buffers for Client ID
* Make sure they are large enough to hold the entire respective string
*
*/
#define BUFFER_SIZE 32 // Reduced that to save space
static char client_id[BUFFER_SIZE];
/*-----*/
/*
* The main MQTT buffers (for MQTT payload)
* We will need to increase if we start publishing more data.
*/
#define APP_BUFFER_SIZE 100
static struct mqtt_connection conn;
static char app_buffer[APP_BUFFER_SIZE];
/*-----*/
static struct mqtt_message *msg_ptr = 0;
static struct etimer publish_periodic_timer;
static struct ctimer ct;
static struct ctimer ct2; // My counter
static char *buf_ptr;
static uint16_t seq_nr_value = 0;
/*-----*/
/* Parent RSSI functionality */
static struct uip_icmp6_echo_reply_notification echo_reply_notification;
static struct etimer echo_request_timer;
static int def_rt_rssi = 0;
/*-----*/
PROCESS(mqtt_demo_process, "MQTT Demo");
/*-----*/
static void
echo_reply_handler(uip_ipaddr_t *source, uint8_t ttl, uint8_t *data,
                  uint16_t datalen)
{
    if(uip_ip6addr_cmp(source, uip_ds6_defrt_choose())) {
        def_rt_rssi = sicslowpan_get_last_rssi();
    }
}
/*-----*/
static void
publish_led_off(void *d)
{
    leds_off(STATUS_LED);
}
/*-----*/
static void
status_led_off(void *d)
{
    leds_off(STATE_LED);
}

```

```

/*-----*/
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
    //printf("Pub Handler: topic='%s' (len=%u), chunk_len=%u\n", topic, topic_len,
    chunk_len);

    /* If we don't like the length, ignore */
    if(topic_len != 8 || chunk_len != 1) {
        //printf("Wrong topic or chunk len > Ignored\n");
        return;
    }

    /* If the tag != leds, ignore */
    if(strncmp(&topic[0], "test", 4) == 0 && strcmp(&topic[5], "one", 3) == 0) {
        if(chunk[0] == '1') {
            leds_on(LED_RED);
        } else if(chunk[0] == '0') {
            leds_off(LED_RED);
        }
        return;
    }
}
/*-----*/
static void
mqtt_event(struct mqtt_connection *m, mqtt_event_t event, void *data)
{
    switch(event) {
    case MQTT_EVENT_CONNECTED: {
        DBG("APP - Application has a MQTT connection\n");
        timer_set(&connection_life, CONNECTION_STABLE_TIME);
        state = STATE_CONNECTED;
        printf("Connected!\n");
        leds_on(STATE_LED);
        ctimer_set(&ct2, 20*PUBLISH_LED_ON_DURATION, status_led_off, NULL);
        break;
    }
    case MQTT_EVENT_DISCONNECTED: {
        DBG("APP - MQTT Disconnect. Reason %u\n", *((mqtt_event_t *)data));

        state = STATE_DISCONNECTED;
        process_poll(&mqtt_demo_process);
        printf("Disconnected :O\n");
        leds_off(STATE_LED);
        break;
    }
    case MQTT_EVENT_PUBLISH: {

```

```

msg_ptr = data;

/* Implement first_flag in publish message? */
if(msg_ptr->first_chunk) {
    msg_ptr->first_chunk = 0;
    printf("*** Received PUB on '%s'. Payload "
           "size: %ib. Content: %s\n\n",
           msg_ptr->topic, msg_ptr->payload_length, msg_ptr->payload_chunk); // ***
DEBUG ONLY !
}

pub_handler(msg_ptr->topic, strlen(msg_ptr->topic), msg_ptr->payload_chunk,
            msg_ptr->payload_length); // Call the pub handler for led action!

break;
}
case MQTT_EVENT_SUBACK: {
    DBG("APP - Application is subscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_UNSUBACK: {
    DBG("APP - Application is unsubscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_PUBACK: {
    DBG("APP - Publishing complete.\n");
    break;
}
default:
    DBG("APP - Application got a unhandled MQTT event: %i\n", event);
    break;
}
}
/*-----*/
static int
construct_client_id(void)
{
    snprintf(client_id, BUFFER_SIZE, "d:%s:%s:%02x%02x%02x%02x%02x%02x",
             DEFAULT_ORG_ID, DEFAULT_TYPE_ID,
             linkaddr_node_addr.u8[0], linkaddr_node_addr.u8[1],
             linkaddr_node_addr.u8[2], linkaddr_node_addr.u8[5],
             linkaddr_node_addr.u8[6], linkaddr_node_addr.u8[7]);

    return 1;
}
/*-----*/
static void
update_config(void)
{

```

```

if(construct_client_id() == 0) {
    /* Fatal error. Client ID larger than the buffer */
    state = STATE_CONFIG_ERROR;
    return;
}
/* Reset the counter */
seq_nr_value = 0;

state = STATE_INIT;

/*
 * Schedule next timer event ASAP
 *
 * If we entered an error state then we won't do anything when it fires.
 *
 * Since the error at this stage is a config error, we will only exit this
 * error state if we get a new config.
 */
etimer_set(&publish_periodic_timer, 0);

return;
}
/*-----*/
static void
subscribe(void)
{
    /* Publish MQTT topic in IBM quickstart format */
    mqtt_status_t status;

    status = mqtt_subscribe(&conn, NULL, DEFAULT_SUBSCRIBE_TOPIC,
MQTT_QOS_LEVEL_0);

    DBG("APP - Subscribing!\n");
    if(status == MQTT_STATUS_OUT_QUEUE_FULL) {
        DBG("APP - Tried to subscribe but command queue was full!\n");
    }
}
/*-----*/
static void
publish(void)
{
    /* Publish MQTT topic in IBM quickstart format */
    int len;
    int remaining = APP_BUFFER_SIZE;
    //int16_t value;

    seq_nr_value++;

    buf_ptr = app_buffer;

```

```

len = snprintf(buf_ptr, remaining,
               "{
               \"d\":{
               \"myName\": \"%s\",
               \"Seq #\":%d,
               \"Uptime (sec)\":%lu",
               BOARD_STRING, seq_nr_value, clock_seconds());

if(len < 0 || len >= remaining) {
    printf("Buffer too short. Have %d, need %d + \\0\\n", remaining, len);
    return;
}

remaining -= len;
buf_ptr += len;

len = snprintf(buf_ptr, remaining, "\",\"RSSI (dBm)\":%d", def_rt_rssi);

if(len < 0 || len >= remaining) {
    printf("Buffer too short. Have %d, need %d + \\0\\n", remaining, len);
    return;
}
remaining -= len;
buf_ptr += len;

len = snprintf(buf_ptr, remaining, "}}");

if(len < 0 || len >= remaining) {
    printf("Buffer too short. Have %d, need %d + \\0\\n", remaining, len);
    return;
}

mqtt_publish(&conn, NULL, DEFAULT_PUBLISH_TOPIC, (uint8_t *)app_buffer,
             strlen(app_buffer), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
DBG("APP - Publish!\\n");
}
/*-----*/
static void
connect_to_broker(void)
{
    /* Connect to MQTT server */
    mqtt_connect(&conn, MQTT_DEMO_BROKER_IP_ADDR,
DEFAULT_BROKER_PORT,
                DEFAULT_PUBLISH_INTERVAL * 3);

    state = STATE_CONNECTING;
}
/*-----*/

```

```

static void
ping_parent(void)
{
    if(uiplib_get_global(ADDR_PREFERRED) == NULL) {
        return;
    }

    uilib_icmp6_send(uiplib_defrt_choose(), ICMP6_ECHO_REQUEST, 0,
                    ECHO_REQ_PAYLOAD_LEN);
}
/*-----*/
static void
state_machine(void)
{
    switch(state) {
    case STATE_INIT:
        /* If we have just been configured register MQTT connection */
        mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,
                    MAX_TCP_SEGMENT_SIZE);

        /*
         * If there is a username and password defined in configuration file
         */
        if(strncasecmp(MQTT_AUTH_USERNAME, "NULL",
strlen(MQTT_AUTH_USERNAME)) != 0 && strlen(MQTT_AUTH_USERNAME) > 1)
        {
            if(strncasecmp(MQTT_AUTH_PASSWORD, "NULL",
strlen(MQTT_AUTH_PASSWORD)) == 0) {
                printf("Username set, but no auth password\n");
                state = STATE_ERROR;
                break;
            } else {
                mqtt_set_username_password(&conn, MQTT_AUTH_USERNAME,
                    MQTT_AUTH_PASSWORD);
                printf("Will authenticate at connection...\n");
            }
        }
    }

    /* _register() will set auto_reconnect. We don't want that. */
    conn.auto_reconnect = 0;
    connect_attempt = 1;

    state = STATE_REGISTERED;
    DBG("Init\n");
    /* Continue */
    case STATE_REGISTERED:
        if(uiplib_get_global(ADDR_PREFERRED) != NULL) {
            /* Registered and with a public IP. Connect */

```

```

    DBG("Registered. Connect attempt %u\n", connect_attempt);
    ping_parent();
    connect_to_broker();
} else {
    leds_on(STATUS_LED);
    ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);
}
etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
return;
break;
case STATE_CONNECTING:
    leds_on(STATUS_LED);
    ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);
    /* Not connected yet. Wait */
    DBG("Connecting (%u)\n", connect_attempt);
    break;
case STATE_CONNECTED:
    /* Continue */
case STATE_PUBLISHING:
    /* If the timer expired, the connection is stable. */
    if(timer_expired(&connection_life)) {
        /*
         * Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
         * attempts if we disconnect after a successful connect
         */
        connect_attempt = 0;
    }

    if(mqtt_ready(&conn) && conn.out_buffer_sent) {
        /* Connected. Publish */
        if(state == STATE_CONNECTED) {
            subscribe();
            state = STATE_PUBLISHING;
        } else {
            leds_on(STATUS_LED);
            ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
            publish();
        }
        etimer_set(&publish_periodic_timer, DEFAULT_PUBLISH_INTERVAL);

        DBG("Publishing\n");
        /* Return here so we don't end up rescheduling the timer */
        return;
    } else {
        /*
         * Our publish timer fired, but some MQTT packet is already in flight
         * (either not sent at all, or sent but not fully ACKd).
         */
    }

```

```

    * This can mean that we have lost connectivity to our broker or that
    * simply there is some network delay. In both cases, we refuse to
    * trigger a new message and we wait for TCP to either ACK the entire
    * packet after retries, or to timeout and notify us.
    */
    DBG("Publishing... (MQTT state=%d, q=%u)\n", conn.state,
        conn.out_queue_full);
}
break;
case STATE_DISCONNECTED:
    DBG("Disconnected\n");
    if(connect_attempt < RECONNECT_ATTEMPTS ||
        RECONNECT_ATTEMPTS == RETRY_FOREVER) {
        /* Disconnect and backoff */
        clock_time_t interval;
        mqtt_disconnect(&conn);
        connect_attempt++;

        interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
            RECONNECT_INTERVAL << 3;

        DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

        etimer_set(&publish_periodic_timer, interval);

        state = STATE_REGISTERED;
        return;
    } else {
        /* Max reconnect attempts reached. Enter error state */
        state = STATE_ERROR;
        DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
    }
    break;
case STATE_CONFIG_ERROR:
    /* Idle away. The only way out is a new config */
    printf("Bad configuration.\n");
    return;
case STATE_ERROR:
default:
    leds_on(STATUS_LED);
    /*
     * 'default' should never happen.
     *
     * If we enter here it's because of some error. Stop timers. The only thing
     * that can bring us out is a new config event
     */
    printf("Default case: State=0x%02x\n", state);
    return;
}

```

```

    /* If we didn't return so far, reschedule ourselves */
    etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
}
/*-----*/
PROCESS_THREAD(mqtt_demo_process, ev, data)
{

    PROCESS_BEGIN();

    printf("-MQTT Client-\n");

    update_config();

    //def_rt_rssi = 0x8000000;
    uip_icmp6_echo_reply_callback_add(&echo_reply_notification,
                                     echo_reply_handler);
    etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);

    /* Main loop */
    while(1) {

        PROCESS_YIELD();

        if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
           ev == PROCESS_EVENT_POLL) {
            state_machine();
        }

        if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {
            ping_parent();
            etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);
        }
    }

    PROCESS_END();
}
/*-----*/

```

➤ MQTT client with AES payload encryption (option 1):

```

/*-----
 *           Author: Sotiris Katsikeas           *
 *           Date: May 2016                       *
 * Version notes: Adjustable AES-ECB encryption on MQTT payload - REDUCED *
 *           Hint: removed multi thread publishing           *
-----*/

#include "contiki-conf.h"
#include "mqtt.h"
#include "net/ipv6/sicslowpan.h"
#include "dev/leds.h"
#include <string.h>
#include "TI_aes.c"
#include "powertrace.h" // Used for powertrace print
/*-----*/
/*
 * Publish to a MQTT broker (e.g. mosquitto) running on the host provided by
MQTT_DEMO_BROKER_IP_ADDR
 */
#define DEFAULT_ORG_ID "mqtt-demo"
/*-----*/
/*
 * A timeout used when waiting for something to happen (e.g. to connect or to
 * disconnect)
 */
#define STATE_MACHINE_PERIODIC (CLOCK_SECOND >> 1)
/*-----*/
/* Provide visible feedback via LEDS during various states */
/* When connecting to broker */
#define CONNECTING_LED_DURATION (CLOCK_SECOND >> 2)

/* Each time we try to publish */
#define PUBLISH_LED_ON_DURATION (CLOCK_SECOND)
/*-----*/
/* Connections and reconnections */
#define RETRY_FOREVER 0xFF
#define RECONNECT_INTERVAL (CLOCK_SECOND * 2)

/*
 * Number of times to try reconnecting to the broker.
 * Can be a limited number (e.g. 3, 10 etc) or can be set to RETRY_FOREVER
 */
#define RECONNECT_ATTEMPTS RETRY_FOREVER
#define CONNECTION_STABLE_TIME (CLOCK_SECOND * 5)
static struct timer connection_life;
static uint8_t connect_attempt;
/*-----*/
/* Various states */

```

```

static uint8_t state;
#define STATE_INIT          0
#define STATE_REGISTERED   1
#define STATE_CONNECTING   2
#define STATE_CONNECTED    3
#define STATE_PUBLISHING   4
#define STATE_DISCONNECTED 5
#define STATE_NEWCONFIG    6
#define STATE_CONFIG_ERROR 0xFE
#define STATE_ERROR        0xFF
/*-----*/
#define RSSI_MEASURE_INTERVAL_MAX 86400 /* secs: 1 day */
#define RSSI_MEASURE_INTERVAL_MIN  5 /* secs */
#define PUBLISH_INTERVAL_MAX      86400 /* secs: 1 day */
#define PUBLISH_INTERVAL_MIN      5 /* secs */
/*-----*/
/* A timeout used when waiting to connect to a network */
#define NET_CONNECT_PERIODIC      (CLOCK_SECOND >> 2)
#define NO_NET_LED_DURATION      (NET_CONNECT_PERIODIC >> 1)
/*-----*/
/* Default configuration values */
#define DEFAULT_TYPE_ID          "cc2420"
#define DEFAULT_PUBLISH_TOPIC    "clients/text"
#define DEFAULT_SUBSCRIBE_TOPIC  "clients/text"
#define DEFAULT_BROKER_PORT      1883
#define DEFAULT_PUBLISH_INTERVAL (CLOCK_SECOND * 5)
#define DEFAULT_KEEP_ALIVE_TIMER 60
#define DEFAULT_RSSI_MEAS_INTERVAL (CLOCK_SECOND * 30)
/*-----*/
/* Payload length of ICMPv6 echo requests used to measure RSSI with def rt */
#define ECHO_REQ_PAYLOAD_LEN     20
/*-----*/
PROCESS_NAME(mqtt_demo_process);
AUTOSTART_PROCESSES(&mqtt_demo_process);
/*-----*/
/* Maximum TCP segment size for outgoing segments of our socket */
#define MAX_TCP_SEGMENT_SIZE    32
/*-----*/
#define STATUS_LED LEDS_GREEN
#define STATE_LED LEDS_BLUE
/*-----*/
/*
 * Buffers for Client ID
 * Make sure they are large enough to hold the entire respective string
 *
 */
#define BUFFER_SIZE 32 // Reduced that to save space
static char client_id[BUFFER_SIZE];
/*-----*/
/*

```

```

* The MQTT connection
*/
static struct mqtt_connection conn;
/*-----*/
static struct mqtt_message *msg_ptr = 0;
static struct etimer publish_periodic_timer;
static struct ctimer ct;
static struct ctimer ct2; // My counter
//static char *buf_ptr;
static uint16_t seq_nr_value = 0;
int subscribeCnt = 0;
/*-----*/
/* Parent RSSI functionality */
static struct uip_icmp6_echo_reply_notification echo_reply_notification;
static struct etimer echo_request_timer;
static int def_rt_rssi = 0;
/*-----*/
/*
* The main AES buffers for encryption and publish
*/
#define NUM_OF_BLOCKS 3 // Number of block to encrypt/decrypt in ECB mode -
SUPPORTS UP TO 4 BUT WORKS UP TO 3 !!!
static char ciphertext[NUM_OF_BLOCKS*16*2+1]; // Ciphertext buffer
static char *buf_ptr2; // Pointer for ciphertext buffer
static int pubCnt = 0;
/*-----*/
// AES encryption symmetric key
unsigned char aes_key[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                          0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
/*-----*/
PROCESS(mqtt_demo_process, "MQTT Demo");
/*-----*/
static void
echo_reply_handler(uip_ipaddr_t *source, uint8_t ttl, uint8_t *data,
                  uint16_t datalen)
{
    if(uip_ip6addr_cmp(source, uip_ds6_defrt_choose())) {
        def_rt_rssi = sicslowpan_get_last_rssi();
    }
}
/*-----*/
static void
publish_led_off(void *d)
{
    leds_off(STATUS_LED);
}
/*-----*/
static void
status_led_off(void *d)

```

```

{
    leds_off(STATE_LED);
}
/*-----*/
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
    //printf("Pub Handler: topic='%s' (len=%u), chunk_len=%u\n", topic, topic_len,
chunk_len);

    // Publish with encrypted payload detected on topic with tag = text
    if(strncmp(&topic[0], "clients", 7) == 0 && strncmp(&topic[8], "text", 4) == 0) {
        //printf("Got encrypted @ topic='%s', chunk='%s' (len=%u)\n", topic, chunk,
chunk_len);
        printf("Got encrypted (%lu)\n", clock_time());
        int i;
        int j = 0;
        // For first 16 bytes
        unsigned char ciphertext[16];
        static char hexstring[4];
        static char *hexstr_ptr;
        hexstr_ptr = hexstring;

        // For second 16 bytes - Comment out for only 16 bytes
        unsigned char ciphertext2[16];
        static char hexstring2[4];
        static char *hexstr_ptr2;
        hexstr_ptr2 = hexstring2;

        // For third 16 bytes - Comment out for only 32 bytes
        unsigned char ciphertext3[16];
        static char hexstring3[4];
        static char *hexstr_ptr3;
        hexstr_ptr3 = hexstring3;

        // For fourth 16 bytes - Comment out for only 48 bytes
        unsigned char ciphertext4[16];
        static char hexstring4[4];
        static char *hexstr_ptr4;
        hexstr_ptr4 = hexstring4;

        for (i=0; i<chunk_len;i+=2){
            if (j<16)
            {
                snprintf(hexstr_ptr, 2, "%c", chunk[i]);
                snprintf(hexstr_ptr+1, 2, "%c", chunk[i+1]);
                int number = (int)strtol(hexstring, NULL, 16); //Convert string hex value to integer
            }
        }
    }
}

```

```

    ciphertext[j] = number;
}
else if (j>=16 && j<32)
{
    snprintf(hexstr_ptr2, 2, "%c", chunk[i]);
    snprintf(hexstr_ptr2+1, 2, "%c", chunk[i+1]);
    int number = (int)strtol(hexstring2, NULL, 16); //Convert string hex value to
integer
    ciphertext2[j-16] = number;
}
else if (j>=32 && j<48)
{
    snprintf(hexstr_ptr3, 2, "%c", chunk[i]);
    snprintf(hexstr_ptr3+1, 2, "%c", chunk[i+1]);
    //printf("\nStr3: %s\n", hexstring3);
    int number = (int)strtol(hexstring3, NULL, 16); //Convert string hex value to
integer
    ciphertext3[j-32] = number;
}
else if (j>=48 && j<64)
{
    snprintf(hexstr_ptr4, 2, "%c", chunk[i]);
    snprintf(hexstr_ptr4+1, 2, "%c", chunk[i+1]);
    //printf("\nStr4: %s\n", hexstring4);
    int number = (int)strtol(hexstring4, NULL, 16); //Convert string hex value to
integer
    ciphertext4[j-48] = number;
}
j++;
}
//ciphertext[chunk_len] = 0; //Null terminate the string
aes_decrypt(ciphertext,aes_key);
printf("Decrypted text:=");
for (i=0; i<16 && ciphertext[i]!=0 ; i++)
    printf("%c",ciphertext[i]);
printf("\n");

// Second 16 bytes
if (NUM_OF_BLOCKS >=2)
{
    aes_decrypt(ciphertext2,aes_key);
    printf("Decrypted text2:=");
    for (i=0; i<16 && ciphertext2[i]!=0 ; i++)
        printf("%c",ciphertext2[i]);
    printf("\n");
}
// Third 16 bytes
if (NUM_OF_BLOCKS >=3)

```

```

    {
        aes_decrypt(ciphertext3,aes_key);
        printf("Decrypted text3:=");
        for (i=0; i<16 && ciphertext3[i]!=0 ; i++)
            printf("%c",ciphertext3[i]);
        printf("\n");
    }
    // Fourth 16 bytes
    if (NUM_OF_BLOCKS == 4)
    {
        aes_decrypt(ciphertext4,aes_key);
        printf("Decrypted text4:=");
        for (i=0; i<16 && ciphertext4[i]!=0 ; i++)
            printf("%c",ciphertext4[i]);
        printf("\n");
    }
    // FOR EVALUATION
    powertrace_print(""); // PRINT ENERGEST AFTER RECEIVE

    mqtt_publish(&conn, NULL, "clients/ack", (uint8_t *)"ACK",
        strlen("ACK"), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
    //END
}
return;
}
/*-----*/
static void
mqtt_event(struct mqtt_connection *m, mqtt_event_t event, void *data)
{
    switch(event) {
    case MQTT_EVENT_CONNECTED: {
        DBG("APP - Application has a MQTT connection\n");
        timer_set(&connection_life, CONNECTION_STABLE_TIME);
        state = STATE_CONNECTED;
        printf("Connected!\n");
        leds_on(STATE_LED);
        ctimer_set(&ct2, 20*PUBLISH_LED_ON_DURATION, status_led_off, NULL);
        break;
    }
    case MQTT_EVENT_DISCONNECTED: {
        DBG("APP - MQTT Disconnect. Reason %u\n", *((mqtt_event_t *)data));

        state = STATE_DISCONNECTED;
        process_poll(&mqtt_demo_process);
        printf("Disconnected :O\n");
        leds_off(STATE_LED);
        break;
    }
}

```

```

case MQTT_EVENT_PUBLISH: {
    msg_ptr = data;

    /* Implement first_flag in publish message? */
    if(msg_ptr->first_chunk) {
        msg_ptr->first_chunk = 0;
        printf("*** Received PUB on '%s'. Payload "
            "size: %ib. Content: %s\n\n",
            msg_ptr->topic, msg_ptr->payload_length, msg_ptr->payload_chunk); // ***
DEBUG ONLY !
    }

    pub_handler(msg_ptr->topic, strlen(msg_ptr->topic), msg_ptr->payload_chunk,
        msg_ptr->payload_length); // Call the pub handler for led action!
    break;
}
case MQTT_EVENT_SUBACK: {
    DBG("APP - Application is subscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_UNSUBACK: {
    DBG("APP - Application is unsubscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_PUBACK: {
    DBG("APP - Publishing complete.\n");
    break;
}
default:
    DBG("APP - Application got a unhandled MQTT event: %i\n", event);
    break;
}
}
/*-----*/
static int
construct_client_id(void)
{
    snprintf(client_id, BUFFER_SIZE, "d:%s:%s:%02x%02x%02x%02x%02x%02x",
        DEFAULT_ORG_ID, DEFAULT_TYPE_ID,
        linkaddr_node_addr.u8[0], linkaddr_node_addr.u8[1],
        linkaddr_node_addr.u8[2], linkaddr_node_addr.u8[5],
        linkaddr_node_addr.u8[6], linkaddr_node_addr.u8[7]);

    return 1;
}
/*-----*/
static void
update_config(void)
{

```

```

if(construct_client_id() == 0) {
    /* Fatal error. Client ID larger than the buffer */
    state = STATE_CONFIG_ERROR;
    return;
}
/* Reset the counter */
seq_nr_value = 0;

state = STATE_INIT;

/*
 * Schedule next timer event ASAP
 *
 * If we entered an error state then we won't do anything when it fires.
 *
 * Since the error at this stage is a config error, we will only exit this
 * error state if we get a new config.
 */
etimer_set(&publish_periodic_timer, 0);

return;
}
/*-----*/
static void
subscribe(char *topic)
{
    /* Subscribe MQTT topic in IBM quickstart format */
    mqtt_status_t status;

    status = mqtt_subscribe(&conn, NULL, topic, MQTT_QOS_LEVEL_0);

    DBG("APP - Subscribing!\n");
    if(status == MQTT_STATUS_OUT_QUEUE_FULL) {
        DBG("APP - Tried to subscribe but command queue was full!\n");
    }
}
/*-----*/
static void
publish(void)
{
    if(1){
        // Publish on subject 'clients/text' in order to test AES

        unsigned char state[16];
        char plaintext[17]; // +1 to hold the string termination char
        static const char text[] = "EncryptedText-#";

        // Buffers for the second 16 bytes - Comment out for only 16 bytes
        unsigned char state2[16];

```

```
char plaintext2[17];
static const char text2[] = "Text2StartsHere2";
```

```
// Buffers for the third 16 bytes - Comment out for only 32 bytes
unsigned char state3[16];
char plaintext3[17];
static const char text3[] = "Text3StartsHere3";
```

```
// Buffers for the fourth 16 bytes - Comment out for only 48 bytes
unsigned char state4[16];
char plaintext4[17];
static const char text4[] = "Text4StartsHere4";
```

```
printf("Encrypting...(%)lu\n", clock_time());
```

```
// First 16 bytes
snprintf(plaintext, sizeof(plaintext), "%s%d", text, pubCnt++%10);
```

```
if (strlen(plaintext)>16)
    printf("ERROR: Plaintext > 16!\n");
// Convert char (plaintext) to hex
int i;
for (i=0; i<16 ;i++){
    if (plaintext[i]!=0)
        state[i] = (int) plaintext[i];
    else
        state[i] = 0x00;
}
```

```
aes_encrypt(state,aes_key);
```

```
// Second 16 bytes - Comment out for only 16 bytes
if (NUM_OF_BLOCKS >=2)
{
    snprintf(plaintext2, sizeof(plaintext2), "%s", text2);
```

```
    if (strlen(plaintext2)>16)
        printf("ERROR: Plaintext2 > 16!\n");
    // Convert char (plaintext) to hex
    for (i=0; i<16 ;i++){
        if (plaintext2[i]!=0)
            state2[i] = (int) plaintext2[i];
        else
            state2[i] = 0x00;
    }
    aes_encrypt(state2,aes_key);
}
```

```
// Third 16 bytes - Comment out for only 32 bytes
if (NUM_OF_BLOCKS >=3)
{
    snprintf(plaintext3, sizeof(plaintext3), "%s", text3);
```

```

if (strlen(plaintext3)>16)
    printf("ERROR: Plaintext3 > 16!\n");
// Convert char (plaintext) to hex
for (i=0; i<16 ;i++){
    if (plaintext3[i]!=0)
        state3[i] = (int) plaintext3[i];
    else
        state3[i] = 0x00;
}
aes_encrypt(state3,aes_key);
}
```

```
// Fourth 16 bytes - Comment out for only 48 bytes
if (NUM_OF_BLOCKS == 4)
{
    snprintf(plaintext4, sizeof(plaintext4), "%s", text4);
```

```

if (strlen(plaintext4)>16)
    printf("ERROR: Plaintext4 > 16!\n");
// Convert char (plaintext) to hex
for (i=0; i<16 ;i++){
    if (plaintext4[i]!=0)
        state4[i] = (int) plaintext4[i];
    else
        state4[i] = 0x00;
}
aes_encrypt(state4,aes_key);
}
```

```
// Now transfer to output buffer
int len;
i = 0;
int j = 0;
int k = 0;
int l = 0;
int remaining = sizeof(ciphertext);
buf_ptr2 = ciphertext;
if (NUM_OF_BLOCKS == 1)
{
    j=16;
    k=16;
    l=16;
}
else if (NUM_OF_BLOCKS == 2)
```

```

{
    k=16;
    l=16;
}
else if (NUM_OF_BLOCKS == 3)
{
    l=16;
}

// Convert the output in a hexadecimal string in order to publish it (1st)
while((i+j+k+l) < (sizeof(state)+sizeof(state2)+sizeof(state3)+sizeof(state4)))
{
    if(i<sizeof(state))
    {
        len = snprintf(buf_ptr2, remaining, "%02x", state[i++]);
        remaining -= len;
        buf_ptr2 += len;
    }
    else if(i>=sizeof(state) && j<sizeof(state2)) // Comment out for only 16 bytes
    {
        len = snprintf(buf_ptr2, remaining, "%02x", state2[j++]);
        remaining -= len;
        buf_ptr2 += len;
    }
    else if(j>=sizeof(state2) && k<sizeof(state3)) // Comment out for only 32 bytes
    {
        len = snprintf(buf_ptr2, remaining, "%02x", state3[k++]);
        remaining -= len;
        buf_ptr2 += len;
    }
    else if(k>=sizeof(state3) && l<sizeof(state4)) // Comment out for only 48 bytes
    {
        len = snprintf(buf_ptr2, remaining, "%02x", state4[l++]);
        remaining -= len;
        buf_ptr2 += len;
        //printf("\nchar:%02x", state4[l-1]);
    }
}
//printf("\nPayload:%s",ciphertext);
mqtt_publish(&conn, NULL, "clients/text", (uint8_t *) ciphertext,
            strlen(ciphertext), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
//printf("Published encrypted!");
// Code for EVALUATION
powertrace_print(""); // PRINT ENERGEST AFTER PUBLISH
//END
}
}
/*-----*/

```

```

static void
connect_to_broker(void)
{
    /* Connect to MQTT server */
    mqtt_connect(&conn, MQTT_DEMO_BROKER_IP_ADDR,
DEFAULT_BROKER_PORT,
                DEFAULT_PUBLISH_INTERVAL * 3);

    state = STATE_CONNECTING;
}
/*-----*/
static void
ping_parent(void)
{
    if(uiplib_get_global(ADDR_PREFERRED) == NULL) {
        return;
    }

    uip_icmp6_send(uiplib_defrt_choose(), ICMP6_ECHO_REQUEST, 0,
                ECHO_REQ_PAYLOAD_LEN);
}
/*-----*/
static void
state_machine(void)
{
    switch(state) {
    case STATE_INIT:
        /* If we have just been configured register MQTT connection */
        mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,
                MAX_TCP_SEGMENT_SIZE);

        /*
         * If there is a username and password defined in configuration file
         */
        if(strncasecmp(MQTT_AUTH_USERNAME, "NULL",
strlen(MQTT_AUTH_USERNAME)) != 0 && strlen(MQTT_AUTH_USERNAME) > 1)
        {
            if(strncasecmp(MQTT_AUTH_PASSWORD, "NULL",
strlen(MQTT_AUTH_PASSWORD)) == 0) {
                printf("Username set, but no auth password\n");
                state = STATE_ERROR;
                break;
            } else {
                mqtt_set_username_password(&conn, MQTT_AUTH_USERNAME,
                MQTT_AUTH_PASSWORD);
                printf("Will authenticate at connection...\n");
            }
        }
    }
}

```

```

/* _register() will set auto_reconnect. We don't want that. */
conn.auto_reconnect = 0;
connect_attempt = 1;

state = STATE_REGISTERED;
DBG("Init\n");
/* Continue */
case STATE_REGISTERED:
if(uiplib_ds6_get_global(ADDR_PREFERRED) != NULL) {
/* Registered and with a public IP. Connect */
DBG("Registered. Connect attempt %u\n", connect_attempt);
ping_parent();
connect_to_broker();
} else {
leds_on(STATUS_LED);
ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);
}
etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
return;
break;
case STATE_CONNECTING:
leds_on(STATUS_LED);
ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);
/* Not connected yet. Wait */
DBG("Connecting (%u)\n", connect_attempt);
break;
case STATE_CONNECTED:
/* Continue */
case STATE_PUBLISHING:
/* If the timer expired, the connection is stable. */
if(timer_expired(&connection_life)) {
/*
* Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
* attempts if we disconnect after a successful connect
*/
connect_attempt = 0;
}

if(mqtt_ready(&conn) && conn.out_buffer_sent) {
/* Connected. Publish */
if(state == STATE_CONNECTED) {
subscribe(DEFAULT_SUBSCRIBE_TOPIC); // Subscribe on default topic
subscribeCnt++;
state = STATE_PUBLISHING;
}
else if (subscribeCnt == 1){
subscribe("leds/red");
}
}

```

```

    subscribeCnt++;
}
else {
    leds_on(STATUS_LED);
    ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
    publish();
}
etimer_set(&publish_periodic_timer, DEFAULT_PUBLISH_INTERVAL);

DBG("Publishing\n");
/* Return here so we don't end up rescheduling the timer */
return;
} else {
    /*
     * Our publish timer fired, but some MQTT packet is already in flight
     * (either not sent at all, or sent but not fully ACKd).
     *
     * This can mean that we have lost connectivity to our broker or that
     * simply there is some network delay. In both cases, we refuse to
     * trigger a new message and we wait for TCP to either ACK the entire
     * packet after retries, or to timeout and notify us.
     */
    DBG("Publishing... (MQTT state=%d, q=%u)\n", conn.state,
        conn.out_queue_full);
}
break;
case STATE_DISCONNECTED:
    DBG("Disconnected\n");
    if(connect_attempt < RECONNECT_ATTEMPTS ||
        RECONNECT_ATTEMPTS == RETRY_FOREVER) {
        /* Disconnect and backoff */
        clock_time_t interval;
        mqtt_disconnect(&conn);
        connect_attempt++;

        interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
            RECONNECT_INTERVAL << 3;

        DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

        etimer_set(&publish_periodic_timer, interval);

        state = STATE_REGISTERED;
        return;
    } else {
        /* Max reconnect attempts reached. Enter error state */
        state = STATE_ERROR;
        DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
    }
}

```

```

    break;
case STATE_CONFIG_ERROR:
    /* Idle away. The only way out is a new config */
    printf("Bad configuration.\n");
    return;
case STATE_ERROR:
default:
    leds_on(STATUS_LED);
    /*
     * 'default' should never happen.
     *
     * If we enter here it's because of some error. Stop timers. The only thing
     * that can bring us out is a new config event
     */
    printf("Default case: State=0x%02x\n", state);
    return;
}

/* If we didn't return so far, reschedule ourselves */
etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
}
/*-----*/
PROCESS_THREAD(mqtt_demo_process, ev, data)
{

    PROCESS_BEGIN();

    printf("-MQTT Client-\n");

    update_config();

    //def_rt_rssi = 0x8000000;
    uip_icmp6_echo_reply_callback_add(&echo_reply_notification,
                                     echo_reply_handler);
    etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);

    // Start powertracing, once every 15 seconds
    //powertrace_start(CLOCK_SECOND * 15);

    /* Main loop */
    while(1) {

        PROCESS_YIELD();

        if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
           ev == PROCESS_EVENT_POLL) {
            state_machine();
        }
    }
}

```

```
if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {
    ping_parent();
    etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);
}
}

PROCESS_END();
}
/*-----*/
```

➤ MQTT client with AES-CBC payload encryption (option 2):

```
/*-----
*           Author: Sotiris Katsikeas           *
*           Date: April 2016                     *
*   Version notes: Includes AES-CBC encryption on MQTT payload - REDUCED *
*   ->PROBLEM: Needs enough free RAM to work continuously. *
*           Hint: removed multi thread publishing
*-----*/
#include "contiki-conf.h"
#include "mqtt.h"
#include "net/ipv6/sicslowpan.h"
#include "dev/leds.h"
#include <string.h>
#include "aes.h"
/*-----*/
/*
 * Publish to a MQTT broker (e.g. mosquitto) running on the host provided by
MQTT_DEMO_BROKER_IP_ADDR
 */
#define DEFAULT_ORG_ID "mqtt-demo"
/*-----*/
/*
 * A timeout used when waiting for something to happen (e.g. to connect or to
 * disconnect)
 */
#define STATE_MACHINE_PERIODIC (CLOCK_SECOND >> 1)
/*-----*/
/* Provide visible feedback via LEDES during various states */
/* When connecting to broker */
#define CONNECTING_LED_DURATION (CLOCK_SECOND >> 2)

/* Each time we try to publish */
#define PUBLISH_LED_ON_DURATION (CLOCK_SECOND)
/*-----*/
/* Connections and reconnections */
#define RETRY_FOREVER 0xFF
#define RECONNECT_INTERVAL (CLOCK_SECOND * 2)

/*
 * Number of times to try reconnecting to the broker.
 * Can be a limited number (e.g. 3, 10 etc) or can be set to RETRY_FOREVER
 */
#define RECONNECT_ATTEMPTS RETRY_FOREVER
#define CONNECTION_STABLE_TIME (CLOCK_SECOND * 5)
static struct timer connection_life;
static uint8_t connect_attempt;
```

```

/*-----*/
/* Various states */
static uint8_t state;
#define STATE_INIT 0
#define STATE_REGISTERED 1
#define STATE_CONNECTING 2
#define STATE_CONNECTED 3
#define STATE_PUBLISHING 4
#define STATE_DISCONNECTED 5
#define STATE_NEWCONFIG 6
#define STATE_CONFIG_ERROR 0xFE
#define STATE_ERROR 0xFF
/*-----*/
#define RSSI_MEASURE_INTERVAL_MAX 86400 /* secs: 1 day */
#define RSSI_MEASURE_INTERVAL_MIN 5 /* secs */
#define PUBLISH_INTERVAL_MAX 86400 /* secs: 1 day */
#define PUBLISH_INTERVAL_MIN 5 /* secs */
/*-----*/
/* A timeout used when waiting to connect to a network */
#define NET_CONNECT_PERIODIC (CLOCK_SECOND >> 2)
#define NO_NET_LED_DURATION (NET_CONNECT_PERIODIC >> 1)
/*-----*/
/* Default configuration values */
#define DEFAULT_TYPE_ID "cc2420"
#define DEFAULT_PUBLISH_TOPIC "clients/text"
#define DEFAULT_SUBSCRIBE_TOPIC "clients/+"
#define DEFAULT_BROKER_PORT 1883
#define DEFAULT_PUBLISH_INTERVAL (CLOCK_SECOND * 20)
#define DEFAULT_KEEP_ALIVE_TIMER 60
#define DEFAULT_RSSI_MEAS_INTERVAL (CLOCK_SECOND * 30)
/*-----*/
/* Payload length of ICMPv6 echo requests used to measure RSSI with def rt */
#define ECHO_REQ_PAYLOAD_LEN 20
/*-----*/
PROCESS_NAME(mqtt_demo_process);
AUTOSTART_PROCESSES(&mqtt_demo_process);
/*-----*/
/* Maximum TCP segment size for outgoing segments of our socket */
#define MAX_TCP_SEGMENT_SIZE 32
/*-----*/
#define STATUS_LED LEDS_GREEN
#define STATE_LED LEDS_BLUE
/*-----*/
/*
 * Buffers for Client ID
 * Make sure they are large enough to hold the entire respective string
 *
 */
#define BUFFER_SIZE 32 // Reduced that to save space
static char client_id[BUFFER_SIZE];

```

```

/*-----*/
/*
 * The MQTT connection
 */
static struct mqtt_connection conn;
/*-----*/
static struct mqtt_message *msg_ptr = 0;
static struct etimer publish_periodic_timer;
static struct ctimer ct;
static struct ctimer ct2; // My counter
static uint16_t seq_nr_value = 0;
int subscribeCnt = 0;
/*-----*/
/* Parent RSSI functionality */
static struct uip_icmp6_echo_reply_notification echo_reply_notification;
static struct etimer echo_request_timer;
static int def_rt_rssi = 0;
/*-----*/
/* Defines for encryption */
#define PRINTUS(text, orig, len) do { \
    printf(text); int i = 0; \
    while(i<len) { printf("%x", orig[i++] & 0xff); } \
    printf("\n"); \
} while(0)
*/
/*-----*/
/*
 * The main buffers for encryption and publish
 */
#define CIPHER_BLOCKS_NUMBER 4 // Increase this for longer message but also more RAM!
static unsigned char *buf_ptr2; // Pointer for ciphertext (in hex format) buffer
static unsigned char app_buff2[2*CIPHER_BLOCKS_NUMBER*N_BLOCK+2];
static uint8_t key1[16] = { 0x00, 0x01, 0x02, 0x03,
                          0x04, 0x05, 0x06, 0x07,
                          0x08, 0x09, 0x0A, 0x0B,
                          0x0C, 0x0D, 0x0E, 0x0F };
static unsigned char iv[N_BLOCK], iv1[N_BLOCK];
static aes_context ctx[1];
/*-----*/
PROCESS(mqtt_demo_process, "MQTT Demo");
/*-----*/
static void
echo_reply_handler(uip_ipaddr_t *source, uint8_t ttl, uint8_t *data,
                  uint16_t datalen)
{
    if(uip_ip6addr_cmp(source, uip_ds6_defrt_choose())) {
        def_rt_rssi = sicslowpan_get_last_rssi();
    }
}

```

```

}
/*-----*/
static void
publish_led_off(void *d)
{
    leds_off(STATUS_LED);
}
/*-----*/
static void
status_led_off(void *d)
{
    leds_off(STATE_LED);
}
/*-----*/
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
    //printf("Pub Handler: topic='%s' (len=%u), chunk len=%u\n", topic, topic_len,
    chunk_len);

    /* If the tag != leds, ignore */
    if(strncmp(&topic[0], "leds", 4) == 0 && strncmp(&topic[5], "red", 3) == 0) {
        if(chunk[0] == '1') {
            leds_on(LED_RED);
        } else if(chunk[0] == '0') {
            leds_off(LED_RED);
        }
    }

    // Publish with encrypted payload detected on topic with tag = text
    else if(strncmp(&topic[0], "clients", 7) == 0 && strncmp(&topic[8], "text", 4) == 0) {

        //printf("Got encrypted @ topic='%s', chunk='%s' (len=%u)\n", topic, chunk,
        chunk_len);

        static unsigned char in[CIPHER_BLOCKS_NUMBER*N_BLOCK],
        res[CIPHER_BLOCKS_NUMBER*N_BLOCK];

        int i;
        static char hexstring[4];
        static char *hexstr_ptr;
        hexstr_ptr = hexstring;
        int j = 0;
        for (i=0; i<chunk_len-1;i+=2){
            sprintf(hexstr_ptr, 2, "%c", chunk[i]);
            sprintf(hexstr_ptr+1, 2, "%c", chunk[i+1]);
            //printf("Str: %s\n", hexstring);
            int number = (int)strtol(hexstring, NULL, 16); //Convert string hex value to integer
            in[j] = number;
            j++;
        }
    }
}

```

```

}
//PRINTU8("in: ", in, sizeof(in));
//printf("#in: %s\n", in);

aes_cbc_decrypt(in, res, sizeof(in)/N_BLOCK, iv1, ctx);
//printf("#decrypt_return: %d\n",c);
//PRINTU8("res: ", res, sizeof(res));
res[CIPHER_BLOCKS_NUMBER*N_BLOCK]=0; // Null terminate the buffer to stop
printf after the plaintext
printf("Plaintext: %s\n", res);
}
return;
}
/*-----*/
static void
mqtt_event(struct mqtt_connection *m, mqtt_event_t event, void *data)
{
    switch(event) {
    case MQTT_EVENT_CONNECTED: {
        DBG("APP - Application has a MQTT connection\n");
        timer_set(&connection_life, CONNECTION_STABLE_TIME);
        state = STATE_CONNECTED;
        printf("Connected!\n");
        leds_on(STATE_LED);
        ctimer_set(&ct2, 20*PUBLISH_LED_ON_DURATION, status_led_off, NULL);
        break;
    }
    case MQTT_EVENT_DISCONNECTED: {
        DBG("APP - MQTT Disconnect. Reason %u\n", *((mqtt_event_t *)data));

        state = STATE_DISCONNECTED;
        process_poll(&mqtt_demo_process);
        printf("Disconnected :O\n");
        leds_off(STATE_LED);
        break;
    }
    case MQTT_EVENT_PUBLISH: {
        msg_ptr = data;

        /* Implement first_flag in publish message? */
        if(msg_ptr->first_chunk) {
            msg_ptr->first_chunk = 0;
            printf("*** Received PUB on '%s'. Payload "
                "size: %ib. Content: %s\n\n",
                msg_ptr->topic, msg_ptr->payload_length, msg_ptr->payload_chunk); // ***
            DEBUG ONLY !
        }
    }
}

```

```

    pub_handler(msg_ptr->topic, strlen(msg_ptr->topic), msg_ptr->payload_chunk,
                msg_ptr->payload_length); // Call the pub handler for led action!
    break;
}
case MQTT_EVENT_SUBACK: {
    DBG("APP - Application is subscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_UNSUBACK: {
    DBG("APP - Application is unsubscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_PUBACK: {
    DBG("APP - Publishing complete.\n");
    break;
}
default:
    DBG("APP - Application got a unhandled MQTT event: %i\n", event);
    break;
}
}
/*-----*/
static int
construct_client_id(void)
{
    snprintf(client_id, BUFFER_SIZE, "d:%s:%s:%02x%02x%02x%02x%02x%02x",
             DEFAULT_ORG_ID, DEFAULT_TYPE_ID,
             linkaddr_node_addr.u8[0], linkaddr_node_addr.u8[1],
             linkaddr_node_addr.u8[2], linkaddr_node_addr.u8[5],
             linkaddr_node_addr.u8[6], linkaddr_node_addr.u8[7]);

    return 1;
}
/*-----*/
static void
update_config(void)
{
    if(construct_client_id() == 0) {
        /* Fatal error. Client ID larger than the buffer */
        state = STATE_CONFIG_ERROR;
        return;
    }
    /* Reset the counter */
    seq_nr_value = 0;

    state = STATE_INIT;

    /*

```

```

* Schedule next timer event ASAP
*
* If we entered an error state then we won't do anything when it fires.
*
* Since the error at this stage is a config error, we will only exit this
* error state if we get a new config.
*/
etimer_set(&publish_periodic_timer, 0);

return;
}
/*-----*/
static void
subscribe(char *topic)
{
/* Subscribe MQTT topic in IBM quickstart format */
mqtt_status_t status;

status = mqtt_subscribe(&conn, NULL, topic, MQTT_QOS_LEVEL_0);

DBG("APP - Subscribing!\n");
if(status == MQTT_STATUS_OUT_QUEUE_FULL) {
DBG("APP - Tried to subscribe but command queue was full!\n");
}
}
/*-----*/
static void
publish(void)
{
/* Publish MQTT */
printf("Encrpyting...\n");

uint8_t data[] = "A message long enough to test encryption and decryption! 1234567";
unsigned char out[CIPHER_BLOCKS_NUMBER*N_BLOCK],
res[CIPHER_BLOCKS_NUMBER*N_BLOCK];
int i=0;

//PRINTU8("Key: ", key1, sizeof(key1));
//PRINTU8("Data: ", data, sizeof(data));
//printf("#Data: %s\n", data);

// Randomly initialize the IV with Contiki's random function
for(i=0;i<N_BLOCK;i++){
iv[i] = random_rand();
iv1[i] = iv[i];
}
aes_cbc_encrypt(data, out, sizeof(data)/N_BLOCK, iv, ctx);
//PRINTU8("out: ", out, sizeof(out));
//printf("#out: %s\n", out);

```

```

int len;
int j = 0;
int remaining = sizeof(app_buff2);
buf_ptr2 = app_buff2;

// Convert the output in a hexadecimal string in order to publish it
while(j<sizeof(out))
{
    len = snprintf(buf_ptr2, remaining, "%02x", out[j++]);
    remaining -= len;
    buf_ptr2 += len;
}

mqtt_publish(&conn, NULL, DEFAULT_PUBLISH_TOPIC, (uint8_t *) app_buff2,
             sizeof(app_buff2), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
printf("Published encrypted!\n");
}
/*-----*/

static void
connect_to_broker(void)
{
    /* Connect to MQTT server */
    mqtt_connect(&conn, MQTT_DEMO_BROKER_IP_ADDR,
                DEFAULT_BROKER_PORT,
                DEFAULT_PUBLISH_INTERVAL * 3);

    state = STATE_CONNECTING;
}
/*-----*/

static void
ping_parent(void)
{
    if(uiplib_get_global(ADDR_PREFERRED) == NULL) {
        return;
    }

    uip_icmp6_send(uip_ds6_defrt_choose(), ICMP6_ECHO_REQUEST, 0,
                  ECHO_REQ_PAYLOAD_LEN);
}
/*-----*/

static void
state_machine(void)
{
    switch(state) {
    case STATE_INIT:
        /* If we have just been configured register MQTT connection */
        mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,

```

```

        MAX_TCP_SEGMENT_SIZE);

/*
 * If there is a username and password defined in configuration file
 */
if(strncasecmp(MQTT_AUTH_USERNAME, "NULL",
strlen(MQTT_AUTH_USERNAME)) != 0 && strlen(MQTT_AUTH_USERNAME) > 1)
{
    if(strncasecmp(MQTT_AUTH_PASSWORD, "NULL",
strlen(MQTT_AUTH_PASSWORD)) == 0) {
        printf("Username set, but no auth password\n");
        state = STATE_ERROR;
        break;
    } else {
        mqtt_set_username_password(&conn, MQTT_AUTH_USERNAME,
MQTT_AUTH_PASSWORD);
        printf("Will authenticate at connection...\n");
    }
}

/* _register() will set auto_reconnect. We don't want that. */
conn.auto_reconnect = 0;
connect_attempt = 1;

state = STATE_REGISTERED;
DBG("Init\n");
/* Continue */
case STATE_REGISTERED:
if(uiplib_ds6_get_global(ADDR_PREFERRED) != NULL) {
    /* Registered and with a public IP. Connect */
    DBG("Registered. Connect attempt %u\n", connect_attempt);
    ping_parent();
    connect_to_broker();
} else {
    leds_on(STATUS_LED);
    ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);
}
etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
return;
break;
case STATE_CONNECTING:
leds_on(STATUS_LED);
ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);
/* Not connected yet. Wait */
DBG("Connecting (%u)\n", connect_attempt);
break;
case STATE_CONNECTED:
/* Continue */

```

```

case STATE_PUBLISHING:
    /* If the timer expired, the connection is stable. */
    if(timer_expired(&connection_life)) {
        /*
         * Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
         * attempts if we disconnect after a successful connect
         */
        connect_attempt = 0;
    }

    if(mqtt_ready(&conn) && conn.out_buffer_sent) {
        /* Connected. Publish */
        if(state == STATE_CONNECTED) {
            subscribe(DEFAULT_SUBSCRIBE_TOPIC); // Subscribe on default topic
            subscribeCnt++;
            state = STATE_PUBLISHING;
        }
        else if (subscribeCnt == 1){
            subscribe("leds/red");
            subscribeCnt++;
        }
        else {
            leds_on(STATUS_LED);
            ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
            publish();
        }
        etimer_set(&publish_periodic_timer, DEFAULT_PUBLISH_INTERVAL);

        DBG("Publishing\n");
        /* Return here so we don't end up rescheduling the timer */
        return;
    } else {
        /*
         * Our publish timer fired, but some MQTT packet is already in flight
         * (either not sent at all, or sent but not fully ACKd).
         *
         * This can mean that we have lost connectivity to our broker or that
         * simply there is some network delay. In both cases, we refuse to
         * trigger a new message and we wait for TCP to either ACK the entire
         * packet after retries, or to timeout and notify us.
         */
        DBG("Publishing... (MQTT state=%d, q=%u)\n", conn.state,
            conn.out_queue_full);
    }
    break;
case STATE_DISCONNECTED:
    DBG("Disconnected\n");
    if(connect_attempt < RECONNECT_ATTEMPTS ||
        RECONNECT_ATTEMPTS == RETRY_FOREVER) {

```

```

    /* Disconnect and backoff */
    clock_time_t interval;
    mqtt_disconnect(&conn);
    connect_attempt++;

    interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
        RECONNECT_INTERVAL << 3;

    DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

    etimer_set(&publish_periodic_timer, interval);

    state = STATE_REGISTERED;
    return;
} else {
    /* Max reconnect attempts reached. Enter error state */
    state = STATE_ERROR;
    DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
}
break;
case STATE_CONFIG_ERROR:
    /* Idle away. The only way out is a new config */
    printf("Bad configuration.\n");
    return;
case STATE_ERROR:
default:
    leds_on(STATUS_LED);
    /*
     * 'default' should never happen.
     *
     * If we enter here it's because of some error. Stop timers. The only thing
     * that can bring us out is a new config event
     */
    printf("Default case: State=0x%02x\n", state);
    return;
}

/* If we didn't return so far, reschedule ourselves */
etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
}
/*-----*/
PROCESS_THREAD(mqtt_demo_process, ev, data)
{

    PROCESS_BEGIN();

    printf("-MQTT Client-\n");

    update_config();

```

```

//def_rt_rssi = 0x8000000;
uip_icmp6_echo_reply_callback_add(&echo_reply_notification,
                                  echo_reply_handler);
etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);

// Initialize the Contiki's random number generator
random_init(518);
// Initialize the AES context and set key
memset(ctx->ksch, '\0', 240); //(N_MAX_ROUNDS + 1) * N_BLOCK =
(14+1)*16=240
ctx->rnd = 0;
aes_set_key(key1,N_BLOCK,ctx);
//printf("#Set key return: %d\n",a);

/* Main loop */
while(1) {

    PROCESS_YIELD();

    if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
        ev == PROCESS_EVENT_POLL) {
        state_machine();
    }

    if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {
        ping_parent();
        etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);
    }
}

PROCESS_END();
}

```

➤ MQTT client with AES-OCB payload encryption – Publishing (encrypting) mote (option 3):

```
/*-----  
*           Author: Sotiris Katsikeas           *  
*           Date: April 2016                     *  
*   Version notes: Includes OCB encryption on MQTT payload - REDUCED   *  
*   Hints: Removed multi thread subscribe and publishing + ONLY PUBLISH (TX) *  
-----*/  
  
#include "contiki-conf.h"  
#include "mqtt.h"  
#include "net/ipv6/sicslowpan.h"  
#include "dev/leds.h"  
#include <string.h>  
#include "ocb.h"  
/*-----*/  
/*  
* Publish to a MQTT broker (e.g. mosquitto) running on the host provided by  
MQTT_DEMO_BROKER_IP_ADDR  
*/  
#define DEFAULT_ORG_ID "mqtt-demo"  
/*-----*/  
/*  
* A timeout used when waiting for something to happen (e.g. to connect or to  
* disconnect)  
*/  
#define STATE_MACHINE_PERIODIC (CLOCK_SECOND >> 1)  
/*-----*/  
/* Provide visible feedback via LEDS during various states */  
/* When connecting to broker */  
#define CONNECTING_LED_DURATION (CLOCK_SECOND >> 2)  
  
/* Each time we try to publish */  
#define PUBLISH_LED_ON_DURATION (CLOCK_SECOND)  
/*-----*/  
/* Connections and reconnections */  
#define RETRY_FOREVER          0xFF  
#define RECONNECT_INTERVAL    (CLOCK_SECOND * 2)  
  
/*  
* Number of times to try reconnecting to the broker.  
* Can be a limited number (e.g. 3, 10 etc) or can be set to RETRY_FOREVER  
*/  
#define RECONNECT_ATTEMPTS     RETRY_FOREVER  
#define CONNECTION_STABLE_TIME (CLOCK_SECOND * 5)  
static struct timer connection_life;  
static uint8_t connect_attempt;  
/*-----*/
```

```

/* Various states */
static uint8_t state;
#define STATE_INIT 0
#define STATE_REGISTERED 1
#define STATE_CONNECTING 2
#define STATE_CONNECTED 3
#define STATE_PUBLISHING 4
#define STATE_DISCONNECTED 5
#define STATE_NEWCONFIG 6
#define STATE_CONFIG_ERROR 0xFE
#define STATE_ERROR 0xFF
/*-----*/
#define RSSI_MEASURE_INTERVAL_MAX 86400 /* secs: 1 day */
#define RSSI_MEASURE_INTERVAL_MIN 5 /* secs */
#define PUBLISH_INTERVAL_MAX 86400 /* secs: 1 day */
#define PUBLISH_INTERVAL_MIN 5 /* secs */
/*-----*/
/* A timeout used when waiting to connect to a network */
#define NET_CONNECT_PERIODIC (CLOCK_SECOND >> 2)
#define NO_NET_LED_DURATION (NET_CONNECT_PERIODIC >> 1)
/*-----*/
/* Default configuration values */
#define DEFAULT_TYPE_ID "cc2420"
#define DEFAULT_PUBLISH_TOPIC "clients/text"
#define DEFAULT_SUBSCRIBE_TOPIC "clients/+"
#define DEFAULT_BROKER_PORT 1883
#define DEFAULT_PUBLISH_INTERVAL (CLOCK_SECOND * 20)
#define DEFAULT_KEEP_ALIVE_TIMER 60
#define DEFAULT_RSSI_MEAS_INTERVAL (CLOCK_SECOND * 30)
/*-----*/
/* Payload length of ICMPv6 echo requests used to measure RSSI with def rt */
#define ECHO_REQ_PAYLOAD_LEN 20
/*-----*/
PROCESS_NAME(mqtt_demo_process);
AUTOSTART_PROCESSES(&mqtt_demo_process);
/*-----*/
/* Maximum TCP segment size for outgoing segments of our socket */
#define MAX_TCP_SEGMENT_SIZE 32
/*-----*/
#define STATUS_LED LEDS_GREEN
#define STATE_LED LEDS_BLUE
/*-----*/
/*
 * Buffers for Client ID
 * Make sure they are large enough to hold the entire respective string
 *
 */
#define BUFFER_SIZE 32 // Reduced that to save space
static char client_id[BUFFER_SIZE];
/*-----*/

```

```

/*
 * The MQTT connection
 */
static struct mqtt_connection conn;
/*-----*/
static struct mqtt_message *msg_ptr = 0;
static struct etimer publish_periodic_timer;
static struct ctimer ct;
static struct ctimer ct2; // My counter
static uint16_t seq_nr_value = 0;
/*-----*/
/* Parent RSSI functionality */
static struct uip_icmp6_echo_reply_notification echo_reply_notification;
static struct etimer echo_request_timer;
static int def_rt_rssi = 0;
/*-----*/
/* Defines for encryption and authentication */
#define HEADER_LEN 16
#define NONCE_LEN 16
#define TAG_LEN 16
#define MAX_TXT_LEN 3*16 // Reduce it to save RAM
#define KEY_LEN 16

#define PRINTU8(text, orig, len) do { \
    printf(text); int i = 0; \
    while(i<len) { printf("%x", orig[i++] & 0xff); } \
    printf("\n"); \
} while(0)

/*-----*/
/*
 * The main buffers for encryption and publish
 */
//static char *buf_ptr2; // Pointer for ciphertext buffer
static unsigned char key[KEY_LEN];
static unsigned char *buf_ptr2; // Pointer for ciphertext+tag (in hex format) buffer
static unsigned char app_buff2[2*MAX_TXT_LEN+2*TAG_LEN+2];
static uint8_t data[MAX_TXT_LEN];
static ocb_state *OCBstate;
static byte ciphertext[MAX_TXT_LEN];
static byte plain[MAX_TXT_LEN];
static byte tag[TAG_LEN];
static byte nonce[NONCE_LEN] = { 0x3F , 0xC4 , 0xE0 , 0xD8 ,
                                0x6A , 0x7B , 0x04 , 0x30 ,
                                0xD8 , 0xCD , 0xB7 , 0x80 ,
                                0x70 , 0xB4 , 0xC5 , 0x5A };
static int pubCnt;
/*-----*/
PROCESS(mqtt_demo_process, "MQTT Demo");

```

```

/*-----*/
static void
echo_reply_handler(uip_ipaddr_t *source, uint8_t ttl, uint8_t *data,
                  uint16_t datalen)
{
    if(uip_ip6addr_cmp(source, uip_ds6_defrt_choose())) {
        def_rt_rssi = sicslowpan_get_last_rssi();
    }
}
/*-----*/
static void
publish_led_off(void *d)
{
    leds_off(STATUS_LED);
}
/*-----*/
static void
status_led_off(void *d)
{
    leds_off(STATE_LED);
}
/*-----*/
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
    // THIS IS INTENTIONALLY LEFT BLANK ;)
}
return;
}
/*-----*/
static void
mqtt_event(struct mqtt_connection *m, mqtt_event_t event, void *data)
{
    switch(event) {
    case MQTT_EVENT_CONNECTED: {
        DBG("APP - Application has a MQTT connection\n");
        timer_set(&connection_life, CONNECTION_STABLE_TIME);
        state = STATE_CONNECTED;
        printf("Connected!\n");
        leds_on(STATE_LED);
        ctimer_set(&ct2, 20*PUBLISH_LED_ON_DURATION, status_led_off, NULL);
        break;
    }
    case MQTT_EVENT_DISCONNECTED: {
        DBG("APP - MQTT Disconnect. Reason %u\n", *((mqtt_event_t *)data));

        state = STATE_DISCONNECTED;
    }
}

```

```

    process_poll(&mqtt_demo_process);
    printf("Disconnected :O\n");
    leds_off(STATE_LED);
    break;
}
case MQTT_EVENT_PUBLISH: {
    msg_ptr = data;

    /* Implement first_flag in publish message? */
    if(msg_ptr->first_chunk) {
        msg_ptr->first_chunk = 0;
        //printf("*** Received PUB on '%s'. Payload "
        //    "size: %ib. Content: %s\n\n",
        //    msg_ptr->topic, msg_ptr->payload_length, msg_ptr->payload_chunk); // ***
    }
    DEBUG ONLY !

    pub_handler(msg_ptr->topic, strlen(msg_ptr->topic), msg_ptr->payload_chunk,
        msg_ptr->payload_length); // Call the pub handler for led action!
    break;
}
case MQTT_EVENT_SUBACK: {
    DBG("APP - Application is subscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_UNSUBACK: {
    DBG("APP - Application is unsubscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_PUBACK: {
    DBG("APP - Publishing complete.\n");
    break;
}
default:
    DBG("APP - Application got a unhandled MQTT event: %i\n", event);
    break;
}
}
/*-----*/
static int
construct_client_id(void)
{
    snprintf(client_id, BUFFER_SIZE, "d:%s:%s:%02x%02x%02x%02x%02x%02x",
        DEFAULT_ORG_ID, DEFAULT_TYPE_ID,
        linkaddr_node_addr.u8[0], linkaddr_node_addr.u8[1],
        linkaddr_node_addr.u8[2], linkaddr_node_addr.u8[5],
        linkaddr_node_addr.u8[6], linkaddr_node_addr.u8[7]);
    return 1;
}

```

```

}
/*-----*/
static void
update_config(void)
{
    if(construct_client_id() == 0) {
        /* Fatal error. Client ID larger than the buffer */
        state = STATE_CONFIG_ERROR;
        return;
    }
    /* Reset the counter */
    seq_nr_value = 0;

    state = STATE_INIT;

    /*
     * Schedule next timer event ASAP
     *
     * If we entered an error state then we won't do anything when it fires.
     *
     * Since the error at this stage is a config error, we will only exit this
     * error state if we get a new config.
     */
    etimer_set(&publish_periodic_timer, 0);

    return;
}
/*-----*/
static void
subscribe(char *topic)
{
    /* Subscribe MQTT topic in IBM quickstart format */
    mqtt_status_t status;

    status = mqtt_subscribe(&conn, NULL, topic, MQTT_QOS_LEVEL_0);

    DBG("APP - Subscribing!\n");
    if(status == MQTT_STATUS_OUT_QUEUE_FULL) {
        DBG("APP - Tried to subscribe but command queue was full!\n");
    }
}
/*-----*/
static void
publish(void)
{
    /* Publish MQTT */
    //printf("Encrypting...\n");

    int i;

```

```

// Initialize the key
for(i=0;i<KEY_LEN;i++){
    key[i] = random_rand();
}
//strcpy(data, "A message long enough to test! 12345678-%");
static const char plaintext[] = "A message long enough to test! 12345678-";
snprintf(data, MAX_TXT_LEN, "%s%d", plaintext, pubCnt++%10);
//PRINTU8("Key: ", key, sizeof(key));
//PRINTU8("Data: ", data, sizeof(data));
//printf("#Data: %s\n", data);

if(!ocb_encrypt(OCBstate,nonce,data,sizeof(data),ciphertext,tag)) printf("ERROR
encrypt\n");
else{
    //PRINTU8("ciphertext: ", ciphertext, sizeof(ciphertext));
    //printf("Ciphertext: %s\n", ciphertext);
}

int len;
int j = 0;
int remaining = sizeof(app_buff2);
buf_ptr2 = app_buff2;
// Convert the Ciphertext in a hexadecimal string in order to publish it
while(j<sizeof(ciphertext))
{
    len = snprintf(buf_ptr2, remaining, "%02x", ciphertext[j++]);
    remaining -= len;
    buf_ptr2 += len;
}

// Convert the Tag in a hexadecimal string in order to publish it
j=0;
while(j<sizeof(tag))
{
    len = snprintf(buf_ptr2, remaining, "%02x", tag[j++]);
    remaining -= len;
    buf_ptr2 += len;
}

mqtt_publish(&conn, NULL, DEFAULT_PUBLISH_TOPIC, (uint8_t *) app_buff2,
            sizeof(app_buff2), MQTT_QOS_LEVEL_0, MQTT_RETAIN_OFF);
printf("Published encrypted!\n");
PRINTU8("ciphertext: ", ciphertext, sizeof(ciphertext));
PRINTU8("Tag: ", tag, sizeof(tag));
}
/*-----*/

static void
connect_to_broker(void)

```

```

{
    /* Connect to MQTT server */
    mqtt_connect(&conn, MQTT_DEMO_BROKER_IP_ADDR,
DEFAULT_BROKER_PORT,
        DEFAULT_PUBLISH_INTERVAL * 3);

    state = STATE_CONNECTING;
}
/*-----*/
static void
ping_parent(void)
{
    if(uiplib_get_global(ADDR_PREFERRED) == NULL) {
        return;
    }

    uip_icmp6_send(uiplib_defrt_choose(), ICMP6_ECHO_REQUEST, 0,
        ECHO_REQ_PAYLOAD_LEN);
}
/*-----*/
static void
state_machine(void)
{
    switch(state) {
    case STATE_INIT:
        /* If we have just been configured register MQTT connection */
        mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,
            MAX_TCP_SEGMENT_SIZE);

        /*
         * If there is a username and password defined in configuration file
         */
        if(strncasecmp(MQTT_AUTH_USERNAME, "NULL",
strlen(MQTT_AUTH_USERNAME)) != 0 && strlen(MQTT_AUTH_USERNAME) > 1)
        {
            if(strncasecmp(MQTT_AUTH_PASSWORD, "NULL",
strlen(MQTT_AUTH_PASSWORD)) == 0) {
                printf("Username set, but no auth password\n");
                state = STATE_ERROR;
                break;
            } else {
                mqtt_set_username_password(&conn, MQTT_AUTH_USERNAME,
                    MQTT_AUTH_PASSWORD);
                printf("Will authenticate at connection...\n");
            }
        }
    }

    /* _register() will set auto_reconnect. We don't want that. */
}

```

```

conn.auto_reconnect = 0;
connect_attempt = 1;

state = STATE_REGISTERED;
DBG("Init\n");
/* Continue */
case STATE_REGISTERED:
if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
    /* Registered and with a public IP. Connect */
    DBG("Registered. Connect attempt %u\n", connect_attempt);
    ping_parent();
    connect_to_broker();
} else {
    leds_on(STATUS_LED);
    ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);
}
etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
return;
break;
case STATE_CONNECTING:
    leds_on(STATUS_LED);
    ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);
    /* Not connected yet. Wait */
    DBG("Connecting (%u)\n", connect_attempt);
    break;
case STATE_CONNECTED:
    /* Continue */
case STATE_PUBLISHING:
    /* If the timer expired, the connection is stable. */
if(timer_expired(&connection_life)) {
    /*
     * Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
     * attempts if we disconnect after a successful connect
     */
    connect_attempt = 0;
}

if(mqtt_ready(&conn) && conn.out_buffer_sent) {
    /* Connected. Publish */
if(state == STATE_CONNECTED) {
        subscribe(DEFAULT_SUBSCRIBE_TOPIC); // Subscribe on default topic
        //subscribeCnt++;
        state = STATE_PUBLISHING;
    }
    /*else if (subscribeCnt == 1){
        subscribe("leds/red");
        subscribeCnt++;
    }*/
else {

```

```

    leds_on(STATUS_LED);
    ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
    publish();
}
etimer_set(&publish_periodic_timer, DEFAULT_PUBLISH_INTERVAL);

DBG("Publishing\n");
/* Return here so we don't end up rescheduling the timer */
return;
} else {
/*
 * Our publish timer fired, but some MQTT packet is already in flight
 * (either not sent at all, or sent but not fully ACKd).
 *
 * This can mean that we have lost connectivity to our broker or that
 * simply there is some network delay. In both cases, we refuse to
 * trigger a new message and we wait for TCP to either ACK the entire
 * packet after retries, or to timeout and notify us.
 */
DBG("Publishing... (MQTT state=%d, q=%u)\n", conn.state,
    conn.out_queue_full);
}
break;
case STATE_DISCONNECTED:
DBG("Disconnected\n");
if(connect_attempt < RECONNECT_ATTEMPTS ||
    RECONNECT_ATTEMPTS == RETRY_FOREVER) {
/* Disconnect and backoff */
clock_time_t interval;
mqtt_disconnect(&conn);
connect_attempt++;

interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
    RECONNECT_INTERVAL << 3;

DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

etimer_set(&publish_periodic_timer, interval);

state = STATE_REGISTERED;
return;
} else {
/* Max reconnect attempts reached. Enter error state */
state = STATE_ERROR;
DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
}
break;
case STATE_CONFIG_ERROR:
/* Idle away. The only way out is a new config */

```

```

    printf("Bad configuration.\n");
    return;
case STATE_ERROR:
default:
    leds_on(STATUS_LED);
    /*
     * 'default' should never happen.
     *
     * If we enter here it's because of some error. Stop timers. The only thing
     * that can bring us out is a new config event
     */
    printf("Default case: State=0x%02x\n", state);
    return;
}

/* If we didn't return so far, reschedule ourselves */
etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
}
/*-----*/
PROCESS_THREAD(mqtt_demo_process, ev, data)
{

    PROCESS_BEGIN();

    printf("-MQTT Client OCB_TX-\n");

    update_config();

    //def_rt_rssi = 0x8000000;
    uip_icmp6_echo_reply_callback_add(&echo_reply_notification,
                                     echo_reply_handler);
    etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);

    // Initialize the Contiki's random number generator and j
    random_init(518);
    int j = 0;
    // Initialize OCB
    OCBstate = ocb_init(key, TAG_LEN, NONCE_LEN, AES256);
    /* if(OCBstate == NULL){
        printf("Can't init OCB!\n\n");
    }*/

    /* Main loop */
    while(1) {

        PROCESS_YIELD();

        if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
           ev == PROCESS_EVENT_POLL) {

```

```
state_machine();
}

if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {
    ping_parent();
    etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);
}
}

PROCESS_END();
}
/*-----*/
```

➤ MQTT client with AES-OCB payload encryption – Receiving (decrypting) mote (option 3):

```

/*-----
*           Author: Sotiris Katsikeas           *
*           Date: April 2016                   *
*   Version notes: Includes OCB encryption on MQTT payload - REDUCED   *
*   Hints: Removed multi thread subscribe and publishing
-----*/
#include "contiki-conf.h"
#include "mqtt.h"
#include "net/ipv6/sicslowpan.h"
#include "dev/leds.h"
#include <string.h>
#include "ocb.h"
/*-----*/
/*
* Publish to a MQTT broker (e.g. mosquitto) running on the host provided by
MQTT_DEMO_BROKER_IP_ADDR
*/
#define DEFAULT_ORG_ID "mqtt-demo"
/*-----*/
/*
* A timeout used when waiting for something to happen (e.g. to connect or to
* disconnect)
*/
#define STATE_MACHINE_PERIODIC (CLOCK_SECOND >> 1)
/*-----*/
/* Provide visible feedback via LEDS during various states */
/* When connecting to broker */
#define CONNECTING_LED_DURATION (CLOCK_SECOND >> 2)

/* Each time we try to publish */
#define PUBLISH_LED_ON_DURATION (CLOCK_SECOND)
/*-----*/
/* Connections and reconnections */
#define RETRY_FOREVER          0xFF
#define RECONNECT_INTERVAL    (CLOCK_SECOND * 2)

/*
* Number of times to try reconnecting to the broker.
* Can be a limited number (e.g. 3, 10 etc) or can be set to RETRY_FOREVER
*/
#define RECONNECT_ATTEMPTS    RETRY_FOREVER
#define CONNECTION_STABLE_TIME (CLOCK_SECOND * 5)
static struct timer connection_life;
static uint8_t connect_attempt;
/*-----*/
/* Various states */

```

```

static uint8_t state;
#define STATE_INIT          0
#define STATE_REGISTERED   1
#define STATE_CONNECTING   2
#define STATE_CONNECTED    3
#define STATE_PUBLISHING   4
#define STATE_DISCONNECTED 5
#define STATE_NEWCONFIG    6
#define STATE_CONFIG_ERROR 0xFE
#define STATE_ERROR        0xFF
/*-----*/
#define RSSI_MEASURE_INTERVAL_MAX 86400 /* secs: 1 day */
#define RSSI_MEASURE_INTERVAL_MIN  5 /* secs */
#define PUBLISH_INTERVAL_MAX      86400 /* secs: 1 day */
#define PUBLISH_INTERVAL_MIN      5 /* secs */
/*-----*/
/* A timeout used when waiting to connect to a network */
#define NET_CONNECT_PERIODIC      (CLOCK_SECOND >> 2)
#define NO_NET_LED_DURATION      (NET_CONNECT_PERIODIC >> 1)
/*-----*/
/* Default configuration values */
#define DEFAULT_TYPE_ID          "cc2420"
#define DEFAULT_PUBLISH_TOPIC    "clients/text"
#define DEFAULT_SUBSCRIBE_TOPIC  "clients/+"
#define DEFAULT_BROKER_PORT      1883
#define DEFAULT_PUBLISH_INTERVAL (CLOCK_SECOND * 20)
#define DEFAULT_KEEP_ALIVE_TIMER 60
#define DEFAULT_RSSI_MEAS_INTERVAL (CLOCK_SECOND * 30)
/*-----*/
/* Payload length of ICMPv6 echo requests used to measure RSSI with def rt */
#define ECHO_REQ_PAYLOAD_LEN     20
/*-----*/
PROCESS_NAME(mqtt_demo_process);
AUTOSTART_PROCESSES(&mqtt_demo_process);
/*-----*/
/* Maximum TCP segment size for outgoing segments of our socket */
#define MAX_TCP_SEGMENT_SIZE    32
/*-----*/
#define STATUS_LED LEDS_GREEN
#define STATE_LED LEDS_BLUE
/*-----*/
/*
 * Buffers for Client ID
 * Make sure they are large enough to hold the entire respective string
 *
 */
#define BUFFER_SIZE 32 // Reduced that to save space
static char client_id[BUFFER_SIZE];
/*-----*/
/*

```

```

* The MQTT connection
*/
static struct mqtt_connection conn;
/*-----*/
static struct mqtt_message *msg_ptr = 0;
static struct etimer publish_periodic_timer;
static struct ctimer ct;
static struct ctimer ct2; // My counter
static uint16_t seq_nr_value = 0;
/*-----*/
/* Parent RSSI functionality */
static struct uip_icmp6_echo_reply_notification echo_reply_notification;
static struct etimer echo_request_timer;
static int def_rt_rssi = 0;
/*-----*/
/* Defines for encryption and authentication */
#define HEADER_LEN 16
#define NONCE_LEN 16
#define TAG_LEN 16
#define MAX_TXT_LEN 3*16 // Reduce it to save RAM
#define KEY_LEN 16

#define PRINTU8(text, orig, len) do { \
    printf(text); int i = 0; \
    while(i<len) { printf("%x", orig[i++] & 0xff); } \
    printf("\n"); \
} while(0)

/*-----*/
/*
* The main buffers for encryption and publish
*/
static unsigned char key[KEY_LEN];
static ocb_state *OCBstate;
static byte ciphertext[MAX_TXT_LEN];
static byte plain[MAX_TXT_LEN];
static byte tag[TAG_LEN];
static byte tag2[TAG_LEN];
static byte nonce[NONCE_LEN] = { 0x3F , 0xC4 , 0xE0 , 0xD8 ,
                                0x6A , 0x7B , 0x04 , 0x30 ,
                                0xD8 , 0xCD , 0xB7 , 0x80 ,
                                0x70 , 0xB4 , 0xC5 , 0x5A };
static int j;
/*-----*/
PROCESS(mqtt_demo_process, "MQTT Demo");
/*-----*/
static void
echo_reply_handler(uip_ipaddr_t *source, uint8_t ttl, uint8_t *data,
                  uint16_t datalen)

```

```

{
    if(uiplib_ip6addr_cmp(source, uip_ds6_defrt_choose())) {
        def_rt_rssi = sicslowpan_get_last_rssi();
    }
}
/*-----*/
static void
publish_led_off(void *d)
{
    leds_off(STATUS_LED);
}
/*-----*/
static void
status_led_off(void *d)
{
    leds_off(STATE_LED);
}
/*-----*/
static void
pub_handler(const char *topic, uint16_t topic_len, const uint8_t *chunk,
            uint16_t chunk_len)
{
    //printf("Pub Handler: topic='%s' (len=%u), chunk_len=%u\n", topic, topic_len,
    chunk_len);

    // Publish with encrypted payload detected on topic with tag = text
    if(strncmp(&topic[0], "clients/text", 12) == 0) {
        printf("Got encrypted on topic='%s', chunk='%s' (length=%u)\n", topic, chunk,
        chunk_len);

        static unsigned char in[MAX_TXT_LEN]; //This has incorrect size! => Tag works
        only this way...
        int i,j;
        static int number;
        static char hexstring[4];
        static char *hexstr_ptr;
        hexstr_ptr = hexstring;
        j=0;
        for (i=0; i<chunk_len-1; i+=2){
            if(j<sizeof(in)+TAG_LEN){ // Prevent massive overflow of in[]
                snprintf(hexstr_ptr, 2, "%c", chunk[i]);
                snprintf(hexstr_ptr+1, 2, "%c", chunk[i+1]);
                number = (int)strtol(hexstring, NULL, 16); //Convert string hex value to integer
                in[j] = number;
                j++;
            }
        }
        //PRINTU8("Ciphertext: ", in, sizeof(in));

```

```

//printf("#Ciphertext: %s\n", in);

ocb_decrypt(OCBstate,nonce,(byte*) in,MAX_TXT_LEN,tag,plain);
//if(!ocb_decrypt(OCBstate,nonce,(byte*) in,sizeof(in),tag,plain)) printf("ERROR
decrypt\n");
//PRINTU8("Decrypted text: ", plain, sizeof(plain));
printf("Decrypted text: %s\n", plain);
PRINTU8("Tag: ", tag, sizeof(tag));

// Evaluation of Tag should be done here (skipped due to limited RAM...)
}
return;
}
/*-----*/
static void
mqtt_event(struct mqtt_connection *m, mqtt_event_t event, void *data)
{
switch(event) {
case MQTT_EVENT_CONNECTED: {
DBG("APP - Application has a MQTT connection\n");
timer_set(&connection_life, CONNECTION_STABLE_TIME);
state = STATE_CONNECTED;
printf("Connected!\n");
leds_on(STATE_LED);
ctimer_set(&ct2, 20*PUBLISH_LED_ON_DURATION, status_led_off, NULL);
break;
}
case MQTT_EVENT_DISCONNECTED: {
DBG("APP - MQTT Disconnect. Reason %u\n", *((mqtt_event_t *)data));

state = STATE_DISCONNECTED;
process_poll(&mqtt_demo_process);
printf("Disconnected :O\n");
leds_off(STATE_LED);
break;
}
case MQTT_EVENT_PUBLISH: {
msg_ptr = data;

/* Implement first_flag in publish message? */
if(msg_ptr->first_chunk) {
msg_ptr->first_chunk = 0;
//printf("*** Received PUB on '%s'. Payload "
// "size: %ib. Content: %s\n\n",
// msg_ptr->topic, msg_ptr->payload_length, msg_ptr->payload_chunk); // ***
DEBUG ONLY !
}
}
}
}

```

```

    pub_handler(msg_ptr->topic, strlen(msg_ptr->topic), msg_ptr->payload_chunk,
                msg_ptr->payload_length); // Call the pub handler for led action!
    break;
}
case MQTT_EVENT_SUBACK: {
    DBG("APP - Application is subscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_UNSUBACK: {
    DBG("APP - Application is unsubscribed to topic successfully\n");
    break;
}
case MQTT_EVENT_PUBACK: {
    DBG("APP - Publishing complete.\n");
    break;
}
default:
    DBG("APP - Application got a unhandled MQTT event: %i\n", event);
    break;
}
}
/*-----*/
static int
construct_client_id(void)
{
    snprintf(client_id, BUFFER_SIZE, "d:%s:%s:%02x%02x%02x%02x%02x%02x",
             DEFAULT_ORG_ID, DEFAULT_TYPE_ID,
             linkaddr_node_addr.u8[0], linkaddr_node_addr.u8[1],
             linkaddr_node_addr.u8[2], linkaddr_node_addr.u8[5],
             linkaddr_node_addr.u8[6], linkaddr_node_addr.u8[7]);

    return 1;
}
/*-----*/
static void
update_config(void)
{
    if(construct_client_id() == 0) {
        /* Fatal error. Client ID larger than the buffer */
        state = STATE_CONFIG_ERROR;
        return;
    }
    /* Reset the counter */
    seq_nr_value = 0;

    state = STATE_INIT;

    /*

```

```

    * Schedule next timer event ASAP
    *
    * If we entered an error state then we won't do anything when it fires.
    *
    * Since the error at this stage is a config error, we will only exit this
    * error state if we get a new config.
    */
etimer_set(&publish_periodic_timer, 0);

    return;
}
/*-----*/
static void
subscribe(char *topic)
{
    /* Subscribe MQTT topic in IBM quickstart format */
    mqtt_status_t status;

    status = mqtt_subscribe(&conn, NULL, topic, MQTT_QOS_LEVEL_0);

    DBG("APP - Subscribing!\n");
    if(status == MQTT_STATUS_OUT_QUEUE_FULL) {
        DBG("APP - Tried to subscribe but command queue was full!\n");
    }
}
/*-----*/
static void
publish(void)
{
    // THIS IS INTENTIONALLY LEFT BLANK ;
}
/*-----*/
static void
connect_to_broker(void)
{
    /* Connect to MQTT server */
    mqtt_connect(&conn, MQTT_DEMO_BROKER_IP_ADDR,
DEFAULT_BROKER_PORT,
        DEFAULT_PUBLISH_INTERVAL * 3);

    state = STATE_CONNECTING;
}
/*-----*/
static void
ping_parent(void)
{
    if(uiplib_get_global(ADDR_PREFERRED) == NULL) {
        return;
    }
}

```

```

    uip_icmp6_send(uip_ds6_defrt_choose(), ICMP6_ECHO_REQUEST, 0,
                  ECHO_REQ_PAYLOAD_LEN);
}
/*-----*/
static void
state_machine(void)
{
    switch(state) {
    case STATE_INIT:
        /* If we have just been configured register MQTT connection */
        mqtt_register(&conn, &mqtt_demo_process, client_id, mqtt_event,
                     MAX_TCP_SEGMENT_SIZE);

        /*
         * If there is a username and password defined in configuration file
         */
        if(strncasecmp(MQTT_AUTH_USERNAME, "NULL",
                      strlen(MQTT_AUTH_USERNAME)) != 0 && strlen(MQTT_AUTH_USERNAME) > 1)
        {
            if(strncasecmp(MQTT_AUTH_PASSWORD, "NULL",
                          strlen(MQTT_AUTH_PASSWORD)) == 0) {
                printf("Username set, but no auth password\n");
                state = STATE_ERROR;
                break;
            } else {
                mqtt_set_username_password(&conn, MQTT_AUTH_USERNAME,
                                           MQTT_AUTH_PASSWORD);
                printf("Will authenticate at connection...\n");
            }
        }

        /* _register() will set auto_reconnect. We don't want that. */
        conn.auto_reconnect = 0;
        connect_attempt = 1;

        state = STATE_REGISTERED;
        DBG("Init\n");
        /* Continue */
    case STATE_REGISTERED:
        if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
            /* Registered and with a public IP. Connect */
            DBG("Registered. Connect attempt %u\n", connect_attempt);
            ping_parent();
            connect_to_broker();
        } else {
            leds_on(STATUS_LED);
            ctimer_set(&ct, NO_NET_LED_DURATION, publish_led_off, NULL);
        }
    }
}

```

```

}
etimer_set(&publish_periodic_timer, NET_CONNECT_PERIODIC);
return;
break;
case STATE_CONNECTING:
leds_on(STATUS_LED);
ctimer_set(&ct, CONNECTING_LED_DURATION, publish_led_off, NULL);
/* Not connected yet. Wait */
DBG("Connecting (%u)\n", connect_attempt);
break;
case STATE_CONNECTED:
/* Continue */
case STATE_PUBLISHING:
/* If the timer expired, the connection is stable. */
if(timer_expired(&connection_life)) {
/*
* Intentionally using 0 here instead of 1: We want RECONNECT_ATTEMPTS
* attempts if we disconnect after a successful connect
*/
connect_attempt = 0;
}

if(mqtt_ready(&conn) && conn.out_buffer_sent) {
/* Connected. Publish */
if(state == STATE_CONNECTED) {
subscribe(DEFAULT_SUBSCRIBE_TOPIC); // Subscribe on default topic
//subscribeCnt++;
state = STATE_PUBLISHING;
}
else {
leds_on(STATUS_LED);
ctimer_set(&ct, PUBLISH_LED_ON_DURATION, publish_led_off, NULL);
publish();
}
etimer_set(&publish_periodic_timer, DEFAULT_PUBLISH_INTERVAL);

DBG("Publishing\n");
/* Return here so we don't end up rescheduling the timer */
return;
} else {
/*
* Our publish timer fired, but some MQTT packet is already in flight
* (either not sent at all, or sent but not fully ACKd).
*
* This can mean that we have lost connectivity to our broker or that
* simply there is some network delay. In both cases, we refuse to
* trigger a new message and we wait for TCP to either ACK the entire
* packet after retries, or to timeout and notify us.
*/

```

```

    DBG("Publishing... (MQTT state=%d, q=%u)\n", conn.state,
        conn.out_queue_full);
}
break;
case STATE_DISCONNECTED:
    DBG("Disconnected\n");
    if(connect_attempt < RECONNECT_ATTEMPTS ||
        RECONNECT_ATTEMPTS == RETRY_FOREVER) {
        /* Disconnect and backoff */
        clock_time_t interval;
        mqtt_disconnect(&conn);
        connect_attempt++;

        interval = connect_attempt < 3 ? RECONNECT_INTERVAL << connect_attempt :
            RECONNECT_INTERVAL << 3;

        DBG("Disconnected. Attempt %u in %lu ticks\n", connect_attempt, interval);

        etimer_set(&publish_periodic_timer, interval);

        state = STATE_REGISTERED;
        return;
    } else {
        /* Max reconnect attempts reached. Enter error state */
        state = STATE_ERROR;
        DBG("Aborting connection after %u attempts\n", connect_attempt - 1);
    }
    break;
case STATE_CONFIG_ERROR:
    /* Idle away. The only way out is a new config */
    printf("Bad configuration.\n");
    return;
case STATE_ERROR:
default:
    leds_on(STATUS_LED);
    /*
     * 'default' should never happen.
     *
     * If we enter here it's because of some error. Stop timers. The only thing
     * that can bring us out is a new config event
     */
    printf("Default case: State=0x%02x\n", state);
    return;
}

/* If we didn't return so far, reschedule ourselves */
etimer_set(&publish_periodic_timer, STATE_MACHINE_PERIODIC);
}

```

```

/*-----*/
PROCESS_THREAD(mqtt_demo_process, ev, data)
{

    PROCESS_BEGIN();

    printf("-MQTT Client OCB_RX-\n");

    update_config();

    //def_rt_rssi = 0x8000000;
    uip_icmp6_echo_reply_callback_add(&echo_reply_notification,
                                     echo_reply_handler);
    etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);

    // Initialize the Contiki's random number generator and j
    random_init(518);
    j = 0;
    // Initialize OCB
    OCBstate = ocb_init(key, TAG_LEN, NONCE_LEN, AES256);
    /* if(OCBstate == NULL){
        printf("Can't init OCB!\n\n");
    }*/

    /* Main loop */
    while(1) {

        PROCESS_YIELD();

        if((ev == PROCESS_EVENT_TIMER && data == &publish_periodic_timer) ||
           ev == PROCESS_EVENT_POLL) {
            state_machine();
        }

        if(ev == PROCESS_EVENT_TIMER && data == &echo_request_timer) {
            ping_parent();
            etimer_set(&echo_request_timer, DEFAULT_RSSI_MEAS_INTERVAL);
        }
    }

    PROCESS_END();
}

```

➤ MQTT client with LLSec (option 4):

In this case, the code of option 0 was used but with the following project-conf.h file parameters:

```
/*-----  
*           Author: Sotiris Katsikeas           *  
*           Date: April 2016                     *  
-----*/  
/**  
* cc2420-mqtt-demo (Zolertia Z1)  
*  
* Project specific configuration defines for the MQTT demo  
*/  
/*-----*/  
#ifndef PROJECT_CONF_H_  
#define PROJECT_CONF_H_  
/*-----*/  
/* User configuration */  
// #define MQTT_DEMO_BROKER_IP_ADDR "fd4a:c00:3660:2:211:32ff:fe26:4119" //  
NAS LUA IPv6 Address  
// #define MQTT_DEMO_BROKER_IP_ADDR "fd4a:0c00:3660:2:7c08:bde2:3fa2:b695" //  
Windows IPv6 Address  
#define MQTT_DEMO_BROKER_IP_ADDR "aaaa::1" // For use with Cooja and NBR  
#define MQTT_AUTH_USERNAME ""  
#define MQTT_AUTH_PASSWORD ""  
#define BOARD_STRING "Sotiris"  
/* Configuration for the network driver */  
#undef NETSTACK_CONF_MAC  
#define NETSTACK_CONF_MAC nullmac_driver  
#undef NETSTACK_CONF_RDC  
// #define NETSTACK_CONF_RDC nullrdc_driver // Use this RDC driver only on  
testing as it keeps radio always ON  
#define NETSTACK_CONF_RDC contikimac_driver // Is the default RDC driver  
/* Enable link layer security! */  
#undef NETSTACK_CONF_LLSEC  
#define NETSTACK_CONF_LLSEC noncoresec_driver  
// #undef NETSTACK_CONF_FRAMER  
// #define NETSTACK_CONF_FRAMER noncoresec_framer  
  
#define LLSEC802154_CONF_SECURITY_LEVEL 6  
  
#undef LLSEC802154_CONF_ENABLED  
#define LLSEC802154_CONF_ENABLED 1  
  
#undef NONCORESEC_CONF_SEC_LVL  
#define NONCORESEC_CONF_SEC_LVL 6
```

```
#define NONCORESEC_CONF_KEY { 0x00 , 0x01 , 0x02 , 0x03 , \  
                               0x04 , 0x05 , 0x06 , 0x07 , \  
                               0x08 , 0x09 , 0x0A , 0x0B , \  
                               0x0C , 0x0D , 0x0E , 0x0F }
```

```
/* END OF LLSEC */
```

```
/*-----*/
```

```
#endif /* PROJECT_CONF_H_ */
```

```
/*-----*/
```