

**Escola Tècnica Superior d'Enginyeria  
Electrònica i Informàtica La Salle**

Treball Final de Màster

Màster Universitari en Enginyeria de Telecomunicació

**IOT CRYPTOGRAPHY SCHEMES COMPARISON**

Adrià Escolano Beltrán

Rosa Maria Alzina Pagès

Julia Sánchez Rodríguez



---

## **ACTA DE L'EXAMEN DEL TREBALL FI DE MASTER**

---

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D. Adrià Escolano Beltrán

va exposar el seu Treball de Fi de Master, el qual va tractar sobre el tema següent:

### **IOT CRYPTOGRAPHY SCHEMES COMPARISON**

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL

VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL



## Abstract

For many years now we have heard about the concept of Internet of Things (**IoT**), however, it seems that it has not been efficiently applied to our daily lives yet. The reasons for this are not unique and, therefore, the complete analysis would be long and cumbersome. Still, one of the undeniably critical points of this process remains in the security of the information.

Most *IoT* solutions are conceived as small and independent networks. In these networks there are relatively few flanks that can be attacked. Instead, taking into account the sustained development that should be presented, these small networks will eventually grow into a giant network. At this point, it will be much more complicated to ensure security which, on the other hand, must be a factor that can never be set aside. Precisely, these security problems must be a key factor in deciding the development of the *IoT*.

As mentioned before, no *IoT* device network can pretend to be connected to the *outside* without being protected against possible attacks, both hardware and software level. We could be thinking about information theft, embezzlement of this information, impersonation, control of devices, etc.

On the same line, within this security layer, the approach to the solution cannot be unique. Taking the CISCO "*Common*" *Cisco IoT Platform Architecture model* as a reference, in this thesis we will focus on the *Embedded Systems and Sensors layer*, where we compare **different cryptographic solutions under the data encryption and transmission framework** (concepts such as identity protection, confidentiality, peer communication, authentication, etc. will be left out of the thesis' scope).

The main issue addressed in this thesis is the fact that the sensors/devices used in IoT networks are small elements; they will use various Operating Systems, CPU types, memory, etc. In addition, many of these units will be (or are) very cheap, with a single mode of operation and a basic network connection that **does not have the power, storage capacity, computation capacity or memory to support the current encryption protocols**. In order to overcome this problem, new schemes capable of running on *IoT* devices, based on current encryption algorithms, must be achieved.

With this goal, we will first study and compare Cryptographic solutions (Divided in Symmetric Key and Asymmetric Key Cryptography) using different algorithms as examples to guide the theoretical study. After that, we will simulate various IoT environments to help us draw conclusions about the usage of encryption techniques in the IoT world.

## Resum

Ja fa un bon grapat d'anys que sentim a parlar sobre el concepte d'*Internet of Things*, tanmateix, sembla que no s'acaba d'aplicar al nostre dia a dia. Els motius d'aquesta no-implantació no són únics i, per tant, l'anàlisi complet seria llarg i feixuc. Tot i això, un dels innegables punts crítics d'aquest procés roman en la seguretat de la informació.

La majoria de les solucions *IoT* es conceben com a xarxes petites i independents. En aquestes xarxes hi ha, relativament, pocs flancs que puguin ser atacats. En canvi, tenint en compte el desenvolupament sostingut que hauria de presentar, aquestes petites xarxes acabaran unint-se en una xarxa gegant. És en aquest punt en el qual serà molt més difícil assegurar la seguretat que, per altra banda, ha de ser un factor que mai es pot deixar de banda. Precisament aquests problemes de seguretat han de ser un factor clau a l'hora de decidir el desenvolupament de l'*IoT*.

Tal i com hem comentat abans, cap xarxa de dispositius *IoT* pot pretendre estar connectada amb l'exterior sense que estigui protegida davant de possibles atacs, ja siguin tant a nivell de hardware com a nivell de software, pensant en furts d'informació, malversació de la mateixa, suplantació d'identitat, control de dispositius, etc.

De la mateixa manera, dins aquesta capa de seguretat, l'enfocament de la solució no pot ser únic. Prenent com a referència el model de CISCO "*Common*" *Cisco IoT Platform Architecture* en aquest treball ens centrarem a la capa *Embedded Systems and Sensors* on compararem diferents solucions criptogràfiques **sota el marc del xifratge i la transmissió d'aquestes dades** (quedaran fora de l'abast conceptes com la *protecció de la identitat, confidencialitat, autenticació dels "participants" de la comunicació, etc.*).

L'interès principal d'aquest treball roman en el fet que els sensors/dispositius utilitzats a les xarxes *IoT* són elements petits, utilitzaran diversos Sistemes Operatius, tipus de CPUs, memòria, etc. A més, moltes d'aquestes unitats seran (o són) molt barates, amb un únic mode de funcionament i una connexió de xarxa bàsica. Això fa que **no tinguin la potència, capacitat d'emmagatzematge, capacitat de càlcul o memòria per a suportar els protocols d'encriptació actuals**. Per tal d'aconseguir superar aquest problema, s'han d'aconseguir nous esquemes que siguin capaços de córrer sobre els dispositius *IoT*, basant-nos en els algorismes actuals de xifratge.

Amb aquest objectiu, primerament estudiarem i compararem solucions criptogràfiques (dividides entre criptografia de clau simètrica i criptografia de clau asimètrica) utilitzant diferents algorismes com a una eina per guiar l'estudi teòric. Després d'això, simularem diversos entorns *IoT* per ajudar-nos a treure conclusions sobre l'ús de les tècniques de xifrat en el món de l'*IoT*.

## Resumen

Hace ya un buen puñado de años que oímos hablar sobre el concepto de Internet of Things, sin embargo, parece que no se acaba de aplicar en nuestro día a día. Los motivos de esta no-implantación no son únicos y, por tanto, el análisis completo sería largo y pesado. Sin embargo, uno de los innegables puntos críticos de este proceso permanece en la seguridad de la información.

La mayoría de las soluciones *IoT* se conciben como redes pequeñas e independientes. En estas redes hay, relativamente, pocos flancos que puedan ser atacados. En cambio, teniendo en cuenta el desarrollo sostenido que debería presentar, estas pequeñas redes acabarán uniéndose en una red gigante. Es en este punto en el que será mucho más difícil asegurar la seguridad de que, por otra parte, debe ser un factor que nunca se puede dejar de lado. Precisamente estos problemas de seguridad deben ser un factor clave en el momento de decidir el desarrollo del *IoT*.

Tal y como hemos comentado antes, ninguna red de dispositivos *IoT* puede pretender estar conectada con el exterior sin que esté protegida ante posibles ataques, ya sean tanto a nivel de hardware como a nivel de software, pensando en hurtos de información, malversación de la misma, suplantación de identidad, control de dispositivos, etc.

Del mismo modo, dentro de esta capa de seguridad, el enfoque de la solución no puede ser único. Tomando como referencia el modelo de CISCO "*Common*" *Cisco IoT Platform Architecture* en este trabajo nos centraremos en la capa *Embedded Systems and Sensors* donde compararemos diferentes soluciones criptográficas bajo el marco del cifrado y la transmisión de estos datos (quedarán fuera del alcance conceptos como la protección de la identidad, confidencialidad, autenticación de los "participantes" de la comunicación, etc.).

El interés principal de este trabajo permanece en el hecho de que los sensores / dispositivos utilizados en las redes *IoT* son elementos pequeños, utilizarán varios Sistemas Operativos, tipo de CPUs, memoria, etc. Además, muchas de estas unidades serán (o son) muy baratas, con un único modo de funcionamiento y una conexión de red básica. Esto hace que **no tengan la potencia, capacidad de almacenamiento, capacidad de cálculo o memoria para soportar los protocolos de encriptación actuales**. Para conseguir superar este problema, se han de conseguir nuevos esquemas que sean capaces de correr sobre los dispositivos IoT, basándonos en los algoritmos actuales de cifrado.

Con este objetivo, primero estudiaremos y compararemos soluciones criptográficas (divididas en criptografía de clave simétrica y criptografía de clave asimétrica) utilizando diferentes algoritmos como herramienta para guiar el estudio teórico. Después de eso, simularemos varios entornos IoT para ayudarnos a sacar conclusiones sobre el uso de las técnicas de cifrado en el mundo de IoT.





Dedicated to my family,  
Thanks for all the support that you have given me during all these years.



## **Acknowledgements**

I would like to thank Julia Sánchez Rodríguez, my project supervisor. Her supervision style has helped me incredibly during the realization of this thesis.

Giving me freedom during the whole project and advising me whenever it was needed to advance firmly through the process, which brought me composure and confidence.



## Table of contents

|   |    |
|---|----|
| Table of contents .....   | 13 |
| List of Figures .....   | 17 |
| List of Tables .....  | 20 |
| Acronyms .....  | 21 |
| 1. Introduction .....   | 23 |
| 1.1. Thesis Context.....  | 23 |
| 1.2. Objectives .....   | 23 |
| 1.3. Work plan .....  | 24 |
| 2. State of the art of the technology used in this thesis ..... | 25 |
| 2.1. Symmetric-Key Cryptography introduction .....              | 25 |
| 2.1.1. Block Ciphers .....                                      | 25 |
| 2.1.2. Stream Ciphers.....                                      | 26 |
| 2.2. Asymmetric Key Cryptography introduction .....             | 26 |
| 2.2.1. Hash function (SHA-3).....                               | 26 |
| 2.2.2. Elliptic Curve Cryptography .....                        | 26 |
| 2.3. CLEFIA (Block Cipher).....                                 | 27 |
| 2.3.1. CLEFIA'S Structure.....                                  | 27 |
| 2.3.2. F-function – Feistel function .....                      | 29 |
| 2.3.3. S-box .....  | 30 |
| 2.3.4. Diffusion Matrices.....                                  | 31 |
| 2.3.5. Hamming weight.....                                      | 31 |
| 2.3.6. DoubleSwap function.....                                 | 32 |
| 2.4. MICKEY v2 (Stream Cipher).....                             | 32 |
| 2.4.1. Input and Output parameters.....                         | 32 |
| 2.4.2. Components of the Keystream generator .....              | 33 |
| 2.4.2.1. The registers.....                                     | 33 |
| 2.4.2.2. Clocking the register R.....                           | 33 |
| 2.4.3. Motivation for the variable clocking.....                | 34 |
| 2.4.4. The S register feedback function .....                   | 34 |
| 2.5. Hash function introduction .....                           | 35 |
| 2.5.1. Basic concepts - Why SHA-3 .....                         | 35 |

|          |  |     |
|----------|--|-----|
| 2.5.2.   | SHA-3 algorithm: KECCAK .....  | 35  |
| 2.5.3.   | Sponge Construction.....   | 36  |
| 2.5.3.1. | Hermetic sponge construction .....   | 37  |
| 2.5.4.   | Bit padding.....   | 38  |
| 2.6.     | Elliptic Curve Cryptography introduction.....                                      | 39  |
| 2.6.1.   | Mathematical resume – Alice and Bob example .....                                  | 39  |
| 2.6.2.   | Elliptic Curve definition.....   | 40  |
| 2.6.3.   | Elliptic Curve Cryptography in IoT.....  | 41  |
| 3.       | Methodology / Project Development.....   | 43  |
| 3.1.     | Keys and Cryptography.....   | 43  |
| 3.2.     | Symmetric Key Cryptography - The ciphers.....                                      | 46  |
| 3.2.1.   | CLEFIA .....   | 47  |
| 3.2.2.   | MICKEY.....  | 51  |
| 3.3.     | Hash function.....   | 57  |
| 3.3.1.   | SHA-3 Algorithm, Hash implementation simulation .....                              | 60  |
| 3.4.     | Elliptic Curve Cryptography - studies and IoT applications.....                    | 64  |
| 4.       | IoT cryptographic algorithms simulation .....                                      | 69  |
| 4.1.     | Simulation premises - AES Standard.....  | 70  |
| 4.2.     | Simulation 1 – Sky mote – Sink-Sender - WSN without cryptographic algorithms.....  | 71  |
| 4.3.     | Simulation 2 – Z1 mote – Sink-Sender - WSN without cryptographic algorithms .....  | 75  |
| 4.4.     | Enabling encryption .....  | 79  |
| 4.5.     | Simulations 3/4 – Sky mote, Z1 mote – Broadcast WSN simulation no encryption ..... | 81  |
| 4.6.     | Simulations 5/6 – Sky mote, Z1 mote – Broadcast WSN simulation encryption .....    | 85  |
| 4.7.     | Simulation 7 – Sky mote – Sink-Sender – WSN with cryptography allowed.....         | 88  |
| 4.8.     | Simulation 8 – Z1 mote – Sink-Sender – WSN with cryptography allowed .....         | 91  |
| 4.9.     | Analysis of the Simulation Results .....   | 94  |
| 4.9.1.   | Broadcast scenario.....  | 94  |
| 4.9.2.   | Sink-Sender environment .....  | 97  |
| 5.       | Results and Conclusions.....   | 99  |
| 5.1.     | Ciphers .....  | 99  |
| 5.2.     | SHA-3 algorithm and Hash study .....   | 101 |
| 5.3.     | Elliptic Curve Cryptography .....  | 102 |
| 5.4.     | Overall conclusions.....   | 102 |

|                              |     |
|------------------------------|-----|
| 6. Future development.....   | 105 |
| 7. Bibliography.....         | 107 |
| 8. Economical study.....     | 111 |
| 9. Appendices.....           | 113 |
| 9.1. Log listener files..... | 113 |
| 9.2. CSC files.....          | 113 |





## List of Figures

**Figure 1:** Gantt Diagram

**Figure 2:** Feistel Function

**Figure 3:** F-Function mathematical definition

**Figure 4:** DoubleSwap function

**Figure 5:** Clocking the R Register with CONTROL\_BIT\_R=0

**Figure 6:** Clocking the R Register with CONTROL\_BIT\_R=1

**Figure 7:** Clocking the S Register

**Figure 8:** Sponge Construction

**Figure 9:** ECC Key computation

**Figure 10:** Elliptic Curve representation

**Figure 11:** Symmetric Key Cryptography

**Figure 12:** Public Key Cryptography

**Figure 13:** Most basic Block Cipher schema

**Figure 14:** Lightweight Cryptography ciphers comparison

**Figure 15:** Gate efficiency comparison

**Figure 16:** Hardware performance: CLEFIA vs PRESENT

**Figure 17:** Stream Cipher scheme

**Figure 18:** Stream Cipher HW performance comparison

**Figure 19:** Compactness summary

**Figure 20:** Throughput summary

**Figure 21:** Power consumption summary

**Figure 22:** Metrics for an output rate of 10 Mbps (estimated typical wireless LAN)

**Figure 23:** Metrics operating at 100kHz clock (low-end RFID/WSN applications)

**Figure 24:** Hash function signing / verification process

**Figure 25:** SHA-3 Performance results under authors ASIC Syntheses

**Figure 26:** Hashing phase 1 - DataHash function

**Figure 27:** Hashing phase 2 - DataHash function

**Figure 28:** Engine structure initialization

**Figure 29:** Hash output format

**Figure 30:** Perfect Hash function example

- Figure 31:** Mathematical display of a Hashing function  $f$
- Figure 32:** Timings for scalar multiplication (normalized for a clock frequency of 7.37MHz)
- Figure 33:** Implementation results, comparison with two other implementations
- Figure 34:** ROM and RAM memory consumption of different implementations
- Figure 35:** Performance for the Key agreement case
- Figure 36:** Performance for the Data Signature case
- Figure 37:** Latency ( $\mu$ s) of the encryption process
- Figure 38:** Throughput information
- Figure 39:** First IoT WSN
- Figure 40:** Client initializations
- Figure 41:** WSN packets transmission
- Figure 42:** Sensor map of the first simulation
- Figure 43:** Mote 3 network isolation
- Figure 44:** Node Info table (I)
- Figure 45:** Node Info table (II)
- Figure 46:** Power distribution per mote
- Figure 47:** Radio Duty Cycle per mote
- Figure 48:** IoT WSN
- Figure 49:** Z1 client initialization
- Figure 50:** Z1 packet transmission
- Figure 51:** Z1 sensor map
- Figure 52:** Node Info table (I)
- Figure 53:** Node Info table (II)
- Figure 54:** Power consumption per mote – Z1
- Figure 55:** Radio Duty Cycle per mote – Z1
- Figure 56:** Broadcast WSN
- Figure 57:** Broadcast WSN simulation packet sending
- Figure 58:** Mote output broadcast data
- Figure 59:** Mote output broadcast power trace data
- Figure 60:** Broadcast WSN, Z1 mote
- Figure 61:** Project\_conf.h file location
- Figure 62:** Project\_conf.h file modification

**Figure 63:** Broadcast WSN, Sky mote, encryption enabled

**Figure 64:** Broadcast WSN, Z1 mote, encryption enabled

**Figure 65:** Sky motes, Sink-sender scenario with cryptography

**Figure 66:** Sink-Sender Simulation going on

**Figure 67:** Sensor map

**Figure 68:** Node Info table (I)

**Figure 69:** Node Info table (II)

**Figure 70:** Average Power Consumption

**Figure 71:** Average Radio Duty Cycle

**Figure 72:** Z1 motes, Sink-Sender scenario with Cryptography

**Figure 73:** Z1 motes, sensor map

**Figure 74:** Node Info table (I)

**Figure 75:** Node Info table (II)

**Figure 76:** Average Power consumption

**Figure 77:** Average Radio Duty Cycle

**Figure 78:** loglistener.txt file portion

## List of Tables

**Table 1:** CLEFIA Design aspects

**Table 2:** S-Box Matrices

**Table 3:** KECCAK technical details

**Table 4:** Ciphers primitive comparison

**Table 5:** CLEFIA Data

**Table 6:** PowerTrace output parameters

## Acronyms

IoT: Internet of Things

NIST: National Institute of Standards and Technology

ECC: Elliptic Curve Cryptography

SHA: Secure Hash Algorithm

WSN: Wireless Sensor Network

ETX: Expected Transmission Count

LPM: Low Power Mode

AES: Advanced Encryption Standard

CCM mode: Mode of operation for cryptographic block ciphers (using 128 bit key size)



# 1. Introduction

## 1.1. Thesis Context

The thesis is developed under the supervision of GRITS department “Research Group on Internet Technologies & Storage”, more precisely, inside the “Cybersecurity” branch; the supervisor of this thesis is the associate professor, Julia Sánchez Rodríguez.

One of the Cybersecurity most important points is its focal presence in the current world. We are living in the world of digital communication and without secure communication systems there is no possible framework for technologies to develop. Every new technology/technological implementation must come with a practical secure schema to be applied; on the contrary, it will become useless, as we cannot benefit from it if anyone could steal information or retrieve data from whatever it is that we are doing.

These concepts undeniably apply the “Internet of Things world”, which is just a small piece of this gear. It seems that the Internet of Things is still not widely spread in our everyday life, many factors may contribute to this but security of information is one of them.

Other researches regarding this topic had been previously carried, the overall sensation is that the limitations and constraints are widely known, but different approaches have been applied to the same problem leaving the topic still opened.

So basically, the main features that differentiate IoT security issues from the traditional ones are the heterogeneous and large-scale objects and networks. These two factors, heterogeneity and complexity, turn the *IoT* security into an issue more difficult to deal with.

Taking these “open” points into account it was interesting to study and compare different current cryptographic solutions in the specific context of the data encryption and transmission, knowing the power, storage capacity, computation capacity or memory constraints that current devices present [i] [vii] [xiii].

Taking all these factors into account, we can see the context where this thesis is going to be developed.

## 1.2. Objectives

The objectives of the IoT crypto-schemes comparison are:

1. Learn about current cryptographic schemas and its functioning (ciphering, deciphering, etc.).
2. Perform a fair comparison in a well-defined framework that would help to determine which schema is the best under the following premises:
  - a. Computational cost
  - b. Data transmission rate
  - c. Battery cost
3. Improve in my ability to research information, filter it, understand it and apply it. Specifically under a scientific point of view. This ability can then be used for any other project or activity in my life, which would require from this knowledge process.

Also, in a more personal point of view, I want to learn about an unknown but interesting subject, which is slightly studied in both the degree and master.

### 1.3. Work plan

The following figure presents the project Gantt Diagram of the master thesis:

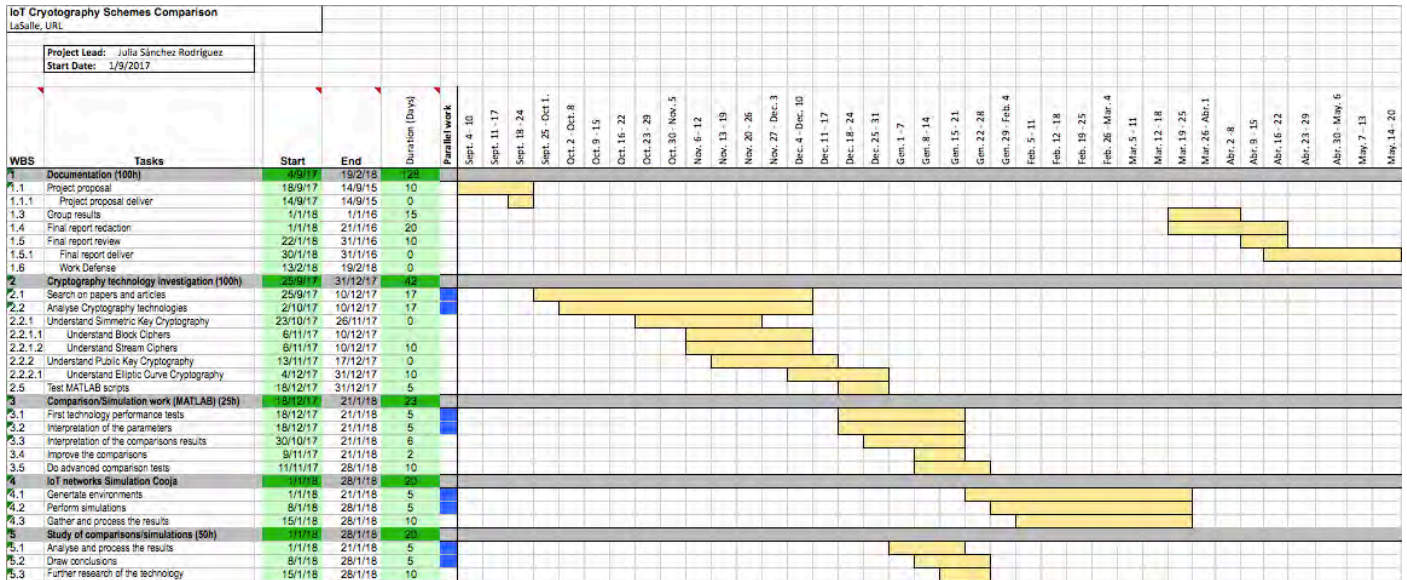


Figure 1: Gantt Diagram

This figure will be further referenced in order to calculate the project costs in section 8.



## 2. State of the art of the technology used in this thesis

As we are living in an era where technological development is constant, the communications field could not be an exception. Indeed, some of the most interesting improvements have been performed in this field. Cybersecurity is not an exception, as it has been investigated hand to hand with these improvements.

In this thesis, as it will be further explained in the Methodology/Project Development point, we will compare the performance of two crypto-schemes:

- **Symmetric Key Cryptography:**
  - Block Cipher
  - Stream Cipher
- **Asymmetric Key Cryptography:**
  - Hash function
  - Elliptic Curve Cryptography

Both under the data encryption and transmission framework, that is taking into account the following parameters when performing the comparison:

- Security
- Flexibility (As the ability to be applied in different systems)
- Performance (Encryption time, transmission speed, etc.)
- Power Consumption
- Price

For the first point, considering the numerous studies that have been carried around [x] [xiii] [xv] [xxi], it will be assumed that all the presented schemes accomplish with the current security standards against various cyber attacks [xxxix].

Firstly, in order to start with any analysis, an introduction to the different concepts/technologies that those cryptographic solutions present is needed. We will present the state of the art of current Block Cipher, Stream Cipher and ECC technologies, plus other necessary concepts (Keccak, Sponge Construction, Feistel Function, etc.) in order to understand the above mention crypto-schemes.

### 2.1. Symmetric-Key Cryptography introduction

In a symmetric-key algorithm both parties use the same key for encryption and decryption.

- Symmetric-key ciphers have high rates of data throughput (Mb/sec) and relatively short keys
- Key must remain secret at both ends and must change frequently
- Many key pairs to be managed in large networks

#### 2.1.1. Block Ciphers

As stated in Masanobu Katagi's and Shiho Moriai's paper [xix], after the Advanced Encryption Standard (AES) was selected, many **block ciphers** with lightweight properties have been proposed. Among them, **CLEFIA** and **PRESENT** are well studied about their security and

implementation. Both algorithms were under consideration in ISO/IEC 29192 “Lightweight Cryptography” and are also ready to use in practical systems.

### 2.1.2. Stream Ciphers

The ECRYPT II eSTREAM project that was held from 2004 to 2008 selected a portfolio of different new stream ciphers. The current eSTREAM portfolio contains 7 algorithms. Among all these algorithms Grain v1, MICKEY v2, and Trivium have lightweight properties, which may be then applied in IoT systems [xv].

## 2.2. Asymmetric Key Cryptography introduction

Asymmetric cryptography algorithms use different keys for encryption and decryption. Each node in the network has a pair of keys, the private key and the public key.

- Only the private key must be kept secret; a private/public key pair may remain unchanged for considerable periods of time, efficient digital signature mechanisms, and smaller number of necessary keys in large networks
- Much slower throughput rates than symmetric-key cryptography and larger key sizes

### 2.2.1. Hash function (SHA-3)

The NIST’s new cryptographic hash algorithm SHA-3 is based on an instance of the KECCAK algorithm that NIST selected as the winner of the SHA-3 Cryptographic Hash Algorithm Competition. The SHA-3 family consists of four cryptographic hash functions and two extendable-output functions. Just as block ciphers can be used to build hash functions, hash functions can be used to build block ciphers. Indeed, **Cryptographic hash functions** are a third type of cryptographic algorithm apart from **Block Ciphers** and **Stream Ciphers**. More precisely they are also considered as part of Public Key Cryptography. [xiv] [xxii]

### 2.2.2. Elliptic Curve Cryptography

While lightweight public key primitives are in demand for key management protocols in smart objects networks, the required resource for public key primitives is much larger than the resources needed in symmetric key primitives so, apparently, it seems that this solution is not the smartest when it comes to *IoT* networks. Still, we will study it, as some researches implement this solution in IoT devices due to the smaller number of necessary keys in large networks.

## 2.3. CLEFIA (Block Cipher)

As a part of the Symmetric Key Cryptography, the CLEFIA block cipher was chosen. It is a proprietary block cipher algorithm, developed by Sony. The block size is 128 bits and the key size can be 128 bits, 192 bits or 256 bits. It is intended to be used in Digital Right Management systems. This block cipher algorithm was chosen due to its open source of information, plus it is one of the algorithms that was under consideration in ISO/IEC 29192 “Lightweight Cryptography” and also ready to use in practical systems. That provided enough amount of information to use it as a pivotal point to study the block ciphers. [xxv]

The design strategy of CLEFIA is to realize good balance on three fundamental directions required for practical ciphers:

- Security
- Speed
- Cost for implementations

### 2.3.1. CLEFIA’S Structure

CLEFIA employs a 4-branch Type-2 generalized Feistel structure. The type-2 Feistel structure has two F-functions in one round for the four data lines case. The type-2 Feistel structure has the following features:

- F-functions are smaller than that of the traditional Feistel structure
- Plural F-functions can be processed simultaneously
- Tends to require more rounds than the traditional Feistel structure

The first feature is a great advantage for software and hardware implementations, and the second one is suitable for efficient implementation especially in hardware. It is concluded that the advantages of the type-2 Feistel structure surpass the disadvantage of the third one for the CLEFIA blockcipher design. Moreover, the new design technique, which is explained next, enables to reduce the number of rounds effectively. [xxv]

**Diffusion Switching Mechanism (DSM)** is one of the novel design approaches of CLEFIA. F-functions employ the Diffusion Switching Mechanism. These F-functions use different diffusion matrices to obtain stronger immunity against differential and linear cryptanalyses. Consequently, the required number of rounds can be reduced.

**Two S-boxes system** CLEFIA employs two different S-boxes based on different algebraic structures, which is expected to increase algebraic immunity.

**Secure and compact key scheduling algorithm.** It is introduced a new key scheduling design. The key scheduling part uses a generalized Feistel structure, and it is possible to share it with the data processing part. Moreover, this structure facilitates easy analysis, and security against related-key attacks is evaluated. The DoubleSwap function used in the key scheduling part has a low cost but also has a good diffusion property. By using the DoubleSwap function, round keys are generated sequentially and efficiently from the intermediate key both in encryption and decryption [xxv].

## Design aspects of CLEFIA

**Table 1:** CLEFIA Design aspects [xxv]

|                             |  |
|-----------------------------|--|
| Generalized Feistel Network | <ul style="list-style-type: none"> <li>· Small size F-functions (32 bit in/out)</li> <li>· No need for the inverse F-functions</li> </ul>  |
| S-Type F-function           | <ul style="list-style-type: none"> <li>· Enabling fast table implementation in software</li> </ul>   |
| DSM                         | <ul style="list-style-type: none"> <li>· Reducing the number of rounds</li> </ul>  |
| S-boxes                     | <ul style="list-style-type: none"> <li>· Very small footprint of <math>S_0</math> and <math>S_1</math> in hardware</li> </ul>  |
| Matrices                    | <ul style="list-style-type: none"> <li>· Using elements with low hamming weights only</li> </ul>   |
| Key Schedule                | <ul style="list-style-type: none"> <li>· Same structure with the data processing part</li> <li>· Only a 128-bit register is required for CLEFIA with 128-bit keys</li> <li>· Small footprint of DoubleSwap function</li> </ul> |

**High efficiency:** CLEFIA was designed to achieve high efficiency in both software and hardware implementations as well as to hold enough security based on the current cryptanalyses. The hardware performance of CLEFIA is particularly advantageous among other block ciphers. In software, CLEFIA with 128-bit keys achieves about 13 cycles/byte, 1.48 Gbps on a 2.4 GHz AMD Athlon 64. This result shows that software performance of CLEFIA is classified into the fastest class among block ciphers. [xxv]. During the next chapter, 3, we will look deeper into various studies to analyze whether these results are effective in different cases or not.

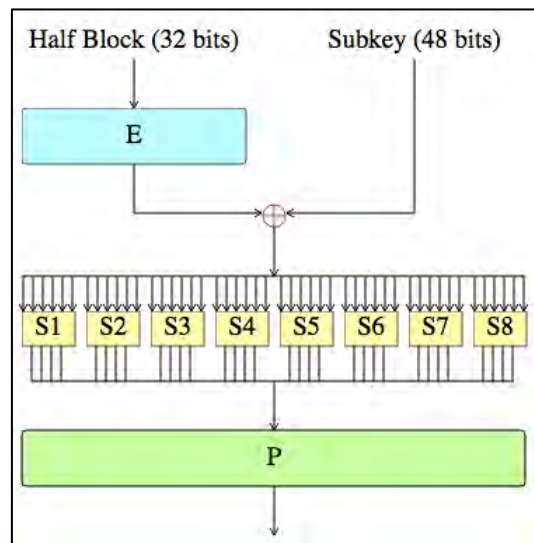
In hardware, an implementation of CLEFIA with 128-bit keys is very small, occupying less than 5K gates by 0.09  $\mu\text{m}$  CMOS ASIC library. This is in the smallest class among block ciphers in the current e-Government (Japan) recommended Ciphers list. For speed-optimized implementations, the performance of CLEFIA achieves 1.6 Gbps with about 6 K gates and 3 Gbps with about 12 K gates. From these results, CLEFIA is unique in hardware efficiency, which is defined by throughput per gate. [xxv]

### 2.3.2. F-function – Feistel function

The F-function, operates on half a block (32 bits) at a time and consists of four stages:

1. Expansion: the 32-bit half-block is expanded to 48 bits using the expansion permutation, denoted  $E$  in the diagram, by duplicating half of the bits. The output consists of eight 6-bit ( $8 * 6 = 48$  bits) pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.
2. Key mixing: the result is combined with a subkey using an  $XOR$  operation. Sixteen 48-bit subkeys—one for each round—are derived from the main key using the key schedule.

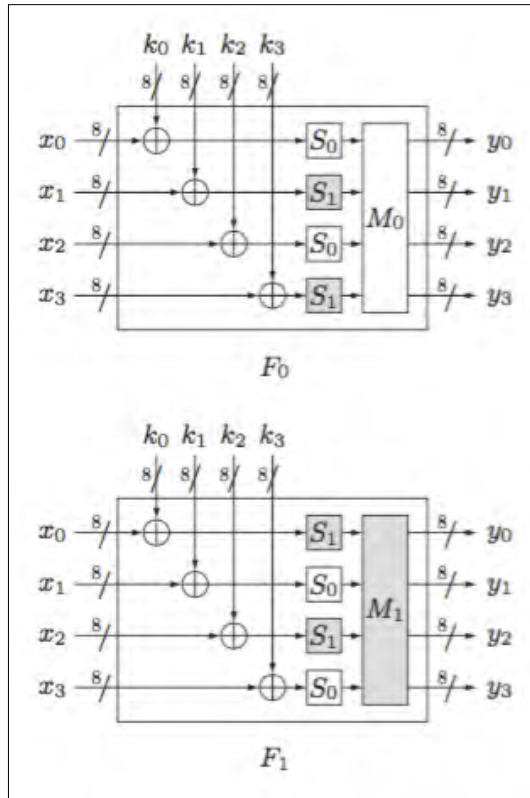
3. Substitution: after mixing in the subkey, the block is divided into eight 6-bit pieces before processing by the S-boxes, or substitution boxes. Each of the eight S-boxes replaces its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table. The S-boxes provide the core of the security of DES—without them; the cipher would be linear, and trivially breakable.



**Figure 2:** Feistel Function [4].

4. Permutation: finally, the 32 outputs from the S-boxes are rearranged according to a fixed permutation, the P-box. This is designed so that, after permutation, the bits from the output of each S-box in this round are spread across four different S-boxes in the next round.

The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides so-called "confusion and diffusion" respectively, a concept identified by Claude Shannon in the 1940s as a necessary condition for a secure yet practical cipher.



In this figure we see the mathematical definition of the two F-functions  $[F_0, F_1]$ . We can see:

- S-Boxes  $[S_0, S_1]$
- Diffusion Matrices  $[M_0, M_1]$

**Figure 3:** F-Function mathematical definition [xxv]

### 2.3.3. S-box

In cryptography, an S-box (substitution-box) is a basic component of symmetric key algorithms, which performs substitution. In block ciphers, they are typically used to obscure the relationship between the key and the ciphertext — Shannon's property of confusion.

In general, an S-box takes some number of input bits,  $m$ , and transforms them into some number of output bits,  $n$ , where  $n$  is not necessarily equal to  $m$ . An  $m \times n$  S-box can be implemented as a lookup table with  $2^m$  words of  $n$  bits each. Fixed tables are normally used, as in the Data Encryption Standard (DES), but in some ciphers the tables are generated dynamically from the key (e.g. the Blowfish and the Twofish encryption algorithms).

One good example of a fixed table is the S-box from DES ( $S_5$ ), mapping 6-bit input into a 4-bit output:

**Table 2:** S-Box Matrix

| $S_5$      | Middle 4 bits of input |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|------------|------------------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
|            | 0000                   | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |      |
| Outer bits | 00                     | 0010 | 1100 | 0100 | 0001 | 0111 | 1010 | 1011 | 0110 | 1000 | 0101 | 0011 | 1110 | 1101 | 0000 | 1110 | 1001 |
|            | 01                     | 1110 | 1011 | 0010 | 1100 | 0100 | 0111 | 1101 | 0001 | 0101 | 0000 | 1111 | 1010 | 0011 | 1001 | 1000 | 0110 |
|            | 10                     | 0100 | 0010 | 0001 | 1011 | 1010 | 1101 | 0111 | 1000 | 1111 | 1001 | 1100 | 0101 | 0110 | 0011 | 0000 | 1110 |
|            | 11                     | 1011 | 1000 | 1100 | 0111 | 0001 | 1110 | 0010 | 1101 | 0110 | 1111 | 0000 | 1001 | 1010 | 0100 | 0101 | 0011 |

Given a 6-bit input, the 4-bit output is found by selecting the row using the outer two bits (the first and last bits), and the column using the inner four bits. For example, an input "011011" has outer bits "01" and inner bits "1101"; the corresponding output would be "1001".

### 2.3.4. Diffusion Matrices

Two matrices  $M_0$  and  $M_1$  used in each F-function, named after the Diffusion Switching Mechanism (DSM). The multiplications of a matrix and a vector are performed in  $GF(2^8)$  defined by the lexicographically first primitive polynomial  $z^8+z^4+z^3+z^2+1$ . Used in the Permutation step [xxv].

$$M_0 = \begin{bmatrix} 0x01 & 0x02 & 0x04 & 0x06 \\ 0x02 & 0x01 & 0x06 & 0x04 \\ 0x04 & 0x06 & 0x01 & 0x02 \\ 0x06 & 0x04 & 0x02 & 0x01 \end{bmatrix}; \quad M_1 = \begin{bmatrix} 0x01 & 0x08 & 0x02 & 0x0a \\ 0x08 & 0x01 & 0x0a & 0x02 \\ 0x02 & 0x0a & 0x01 & 0x08 \\ 0x0a & 0x02 & 0x08 & 0x01 \end{bmatrix}$$

### 2.3.5. Hamming weight

The Hamming weight of a string is the number of symbols that are different from the zero-symbol of the alphabet used. It is thus equivalent to the Hamming distance from the all-zero string of the same length. For the most typical case, a string of bits, this is the number of 1's in the string, or the digit sum of the binary representation of a given number and the  $\ell_1$  norm of a bit vector. In this binary case, it is also called the population count, popcount, sideways sum, or bit summation. Hamming weight calculation can be seen in the following example:

| String   | Hamming weight |
|----------|----------------|
| 11101    | 4              |
| 11101001 | 5              |

### 2.3.6. DoubleSwap function

A simple linear operation called DoubleSwap is used in the key scheduling part. The DoubleSwap function is a bit operation taking a 128-bit value as an input, then dividing the input into 4 parts and shuffling them to increase security. The DoubleSwap function can be used for rapidly generating round keys while maintaining the efficiency of software and hardware implementation.

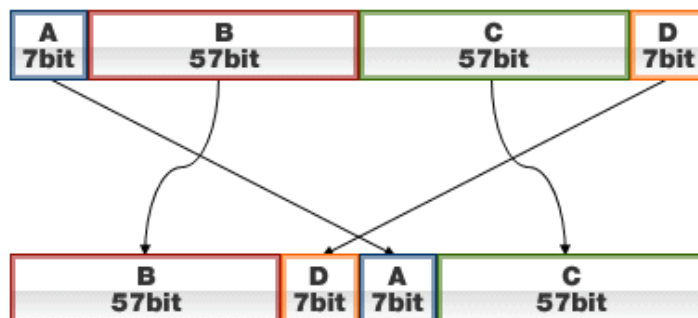


Figure 4: DoubleSwap Function [xxxi].

## 2.4. MICKEY v2 (Stream Cipher)

In cryptography, Mutual Irregular Clocking KEYstream generator (MICKEY) is a stream cipher algorithm developed by Steve Babbage and Matthew Dodd. The cipher is designed to be used in hardware platforms with limited resources, that is why is interesting to study in this thesis. Also, it was one of the three ciphers accepted into Profile 2 of the eSTREAM portfolio, providing us with enough amount of information to investigate and compare. Finally, the algorithm is not patented and is free for any use. These three properties were the reason to choose this algorithm as the pivotal point to study the stream ciphers. [xx]

The algorithm is based around two registers R and S, each of which has two modes of clocking selected by a control bit. With this as a starting point, it was lead to design a clocking rule for the ensemble (R, S), in which the control bit for each register is formed from combination of bits dependent on both registers.

It was also intended from the outset that R should clock as a Galois-stepping feedback shift register either 1 or J times, given that J steps can be implemented efficiently in a single clock cycle. The register S, on the other hand, was intended to clock non-linearly, in two different ways. Successive bits of keystream are formed by combining bits from the registers R and S. Broadly speaking, the idea was that the linearity of R would ensure good statistical properties and guarantees about period, whilst the non-linearity of S would protect against attacks that might be mounted against a linear system. [xx]

### 2.4.1. Input and Output parameters

As it was said before, MICKEY 2.0 is aimed at resource-constrained hardware platforms. That is why this stream cipher is intended to have low complexity in hardware, while providing a high level of security.



MICKEY 2.0 takes two input parameters:

- An 80-bit secret key  $K$ , whose bits are labeled  $k_0, \dots, k_{79}$  ;
- An initialization variable  $IV$ , anywhere between 0 and 80 bits in length, whose bits are labeled  $iv_0, \dots, iv_{IVLENGTH-1}$ .

The *keystream* bits output by MICKEY 2.0 are labeled  $z_0, z_1, \dots$ . Ciphertext is produced from plaintext by bitwise XOR with *keystream* bits, as in most stream ciphers.

The maximum length of *keystream* sequence that may be generated with a single  $(K, IV)$  pair is  $2^{40}$  bits. It is acceptable to generate  $2^{40}$  such sequences, all from the same  $K$  but with different values of  $IV$ . It is not acceptable to use two initialization variables of different lengths with the same  $K$ . And it is not, of course, acceptable to reuse the same value of  $IV$  with the same  $K$ . [xx]

## 2.4.2. Components of the Keystream generator

### 2.4.2.1. The registers

The generator is built from two registers  $R$  and  $S$ . Each register is 100 stages long, each stage containing one bit. It is labeled the bits in the registers  $r_0 \dots r_{99}$  and  $s_0 \dots s_{99}$  respectively. Broadly speaking,  $R$  is thought of as “the linear register” and  $S$  as “the non-linear register”.

### 2.4.2.2. Clocking the register R

Register  $R$  has a set of feedback taps  $RTAPS$ , and clocks in one of two ways according to the value of a control bit  $CONTROL\_BIT\_R$ . When the value of  $CONTROL\_BIT\_R = 0$ , the clocking of  $R$  is a standard linear feedback shift register clocking operation

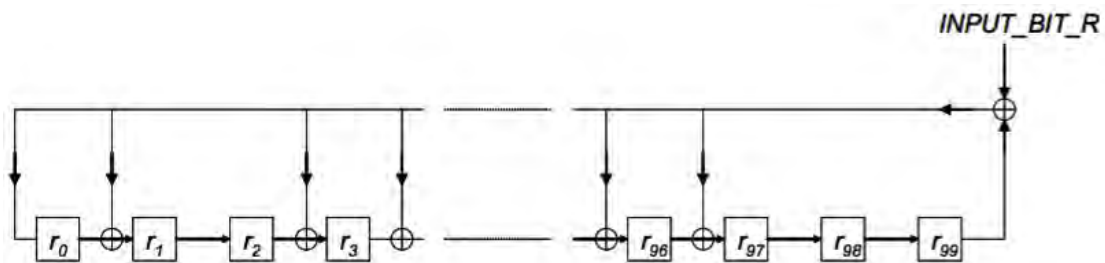


Figure 5: Clocking the R Register with  $CONTROL\_BIT\_R=0$  [xx]

When  $CONTROL\_BIT = 1$ , as well as shifting each bit in the register to the right, it is also XORed it back into the current stage, as shown in Figure 6. This corresponds to multiplication by  $x + 1$  in the same field.

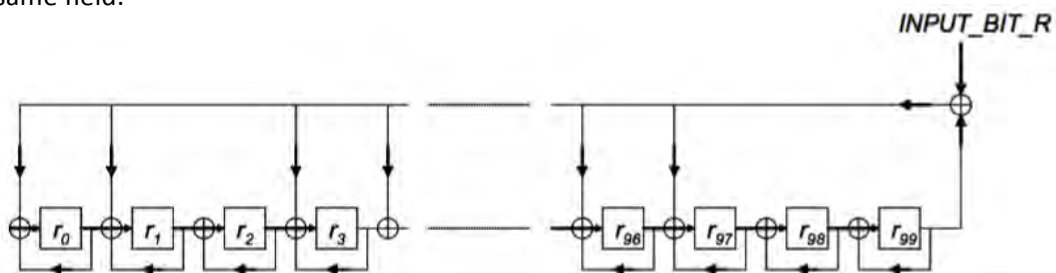


Figure 6: Clocking the R Register with  $CONTROL\_BIT\_R=1$  [xx]

### 2.4.3. Motivation for the variable clocking

Stream ciphers making use of variable clocking often lend themselves to statistical attacks, in which the attacker guesses how many times the register has been clocked at a particular time. There are a number of characteristics of a cipher design that may make such attacks possible. Attacks based on guessing a likely number of clocks of the 89-stage register may be possible.

The principles behind the design of the MICKEY algorithms are:

- To take all of the benefits of variable clocking, in protecting against many forms of attack;
- To guarantee period and local randomness;
- Subject to those, to reduce the susceptibility to statistical attacks as far as possible.

In the MICKEY family of stream ciphers, the register R acts as the ‘engine’, ensuring that the state of the generator does not repeat within the generation of a single keystream sequence, and ensuring good local statistical properties. The influence of R on the clocking of S also prevents S from becoming stuck in a short cycle. [xx]

### 2.4.4. The S register feedback function

For any fixed value of CONTROL\_BIT\_S, the clocking function of S is invertible (so that the space of possible register values is not reduced by clocking S). Our design goal for the clocking function of S can be stated as follows. Assume that the initial state of S is randomly selected, and that the sequence of values of CONTROL BIT S applied to the clocking of S are also randomly selected. Then consider the sequence  $(s_0(i))_{i=0,1,2,\dots}$ . (By  $s_0(i)$  meaning the contents of  $s_0$  after the generator has been clocked  $i$  times.) It is desired to avoid any strong affine relations in that sequence — that is, it is not wanted there to exist a set I such that the value. [xx]

$p = \sum_{i \in I} s_0(i)$  is especially likely to be equal to 0 (or to 1) as the initial state and CONTROL\_BIT\_S range over all possible values.

The reason for this design goal is to avoid attacks based on establishing a probabilistic linear model (i.e. a set I as described above) that would allow a linear combination of keystream bits to be strongly correlated to a combination of bits only from the (‘linear’, ‘weaker’) R register. [xx]

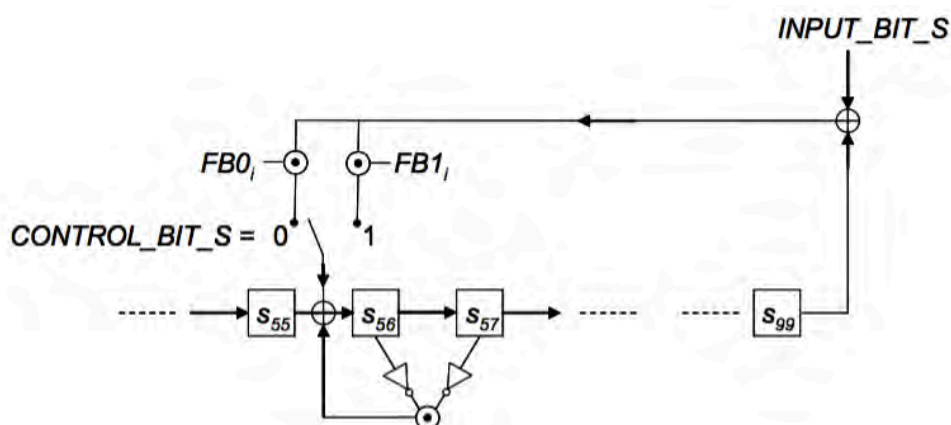


Figure 7: Clocking the S Register [xx]

## 2.5. Hash function introduction

A cryptographic hash algorithm (alternatively, hash "function") is designed to provide a random mapping from a string of binary data to a fixed-size "message digest" and achieve certain security properties. Hash algorithms can be used for digital signatures, message authentication codes, key derivation functions, pseudo random functions, and many other security applications.

### 2.5.1. Basic concepts - Why SHA-3

In 2004-2005, several cryptographic hash algorithms were successfully attacked, and serious attacks were published against the NIST-approved SHA-1. In response, NIST held two public workshops to assess the status of its approved hash algorithms, and to solicit public input on its cryptographic hash algorithm policy and standard. As a result of these workshops, NIST decided to develop a new cryptographic hash algorithm for standardization through a public competition. The new hash algorithm would be referred to as SHA-3 (Secure Hash Algorithm-3). Competition ended on October 2, 2012, and **KECCAK** algorithm was announced as the winning algorithm to be standardized as the new SHA-3. [iv] [vii]

### 2.5.2. SHA-3 algorithm: KECCAK

From the creators' webpage (<https://keccak.team/keccak.html>): "Keccak is a versatile cryptographic function. Best known as a hash function, it nevertheless can also be used for authentication, (authenticated) encryption and pseudo-random number generation. Its structure is the extremely simple sponge construction and internally it uses the innovative Keccak-f cryptographic permutation."

After its selection as the winner of the SHA-3 competition, **KECCAK** has been standardized in 3GPP TS 35.231 for mobile telephony (TUAK), and in NIST standards FIPS 202 and SP 800-185. As a consequence, it has received extensive public scrutiny and third-party cryptanalysis.

In the following table we present the standard technical details:

**Table 3:** KECCAK technical details [xlili]

|                         |   |
|-------------------------|---|
| <b>Synopsis</b>         | The KECCAK sponge functions   |
| <b>Designed by</b>      | Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche   |
| <b>Implements</b>       | An extendable-output function (XOF), i.e., the generalization of a cryptographic hash function with arbitrary output length   |
| <b>Construction</b>     | The sponge construction   |
| <b>Primitive</b>        | One of the KECCAK-f[b] permutations, where b is 25, 50, 100, 200, 400, 800 or 1600 bits. In the scope of the FIPS 202 and SP 800-185 standards, the largest permutation KECCAK-f[1600] is used. Nevertheless, smaller (or more "lightweight") permutations can be used in constrained environments. |
| <b>Parameterized by</b> | The capacity c and by the bitrate r   |
| <b>Instances</b>        | The instances are denoted KECCAK[r, c]. The capacity c determines the proven  |

|               |  |
|---------------|--|
|               | security strength against generic attacks, i.e., for a security level of $n$ bits, the capacity must be $c=2n$ . When summed, $r+c$ must be the width of the permutation among 25, 50, 100, 200, 400, 800 and 1600 bits. |
| <b>Status</b> | Winner of the SHA-3 competition, standardized in 3GPP TS 35.231, FIPS 202 and SP 800-185   |

### 2.5.3. Sponge Construction

In the context of cryptography, the **sponge construction** is a mode of operation, based on a fixed-length permutation (or transformation) and on a padding rule, which builds a function mapping variable-length input to variable-length output. Such a function is called a sponge function. It takes as input an element of  $Z_2^*$ , i.e., a binary string of any length, and returns a binary string with any requested length, i.e., an element of  $Z_2^n$ , with  $n$  a user-supplied value.

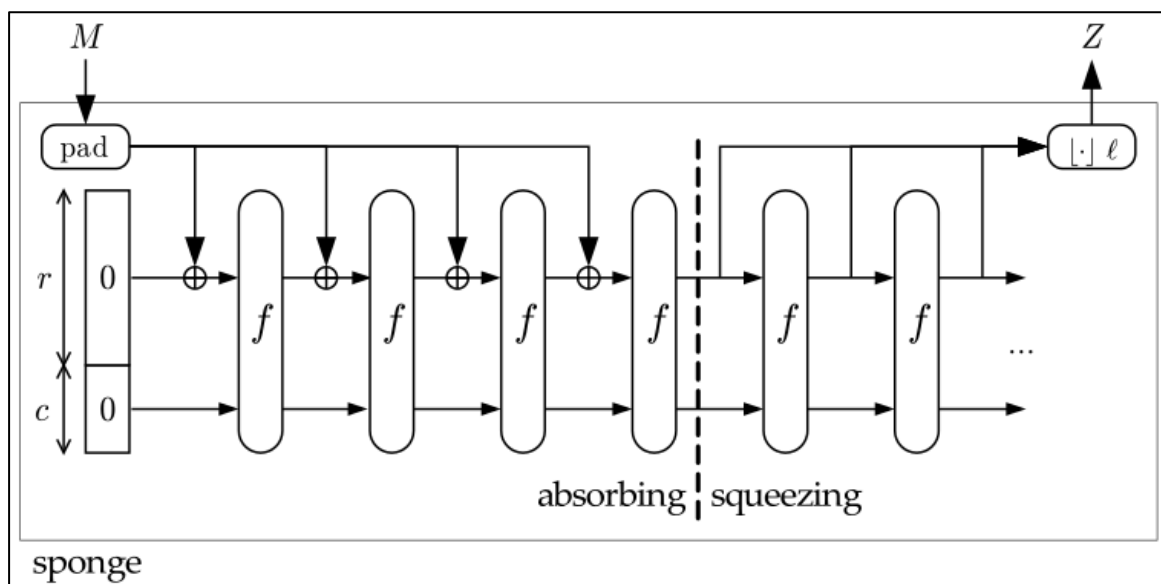
A sponge function is a generalization of both hash functions, which have a fixed output length, and stream ciphers, which have a fixed input length. It operates on a finite state by iteratively applying the inner permutation to it, interleaved with the entry of input or the retrieval of output [xlirii].

Original motivation for introducing the sponge construction was to serve as a reference for expressing security claims. Protocols or modes of hash functions are often proven secure in the random oracle model, i.e., assuming the hash function is a random oracle. *The random oracle is an abstract primitive with the best possible cryptographic properties: it returns for every possible query a random (infinite) string.* In practice, concrete hash functions are used instead and hence the security requirement they must satisfy is to behave like a random oracle. All known practical hash functions operate in an iterated way using finite memory. This may give rise to internal collisions: different inputs leading to the same internal state and consequently to the same output. Alternatively, applying the sponge construction with a random permutation results in a so-called random sponge. It turns out that a random sponge is as strong as a random oracle, except for the effects induced by the finite memory. Random sponges are thus well suited to replace random oracles for expressing security claims [xlirii].

Additionally, the sponge construction can be used to implement a wide spectrum of symmetric cryptography functions. This includes hashing, reseedable pseudo random bit sequence generation, key derivation, encryption, message authentication code (MAC) computation and authenticated encryption. The fundamental cryptographic primitive underlying all this is a fixed-length permutation. These permutation-based modes form efficient alternatives for the current block-cipher dominated cryptographic practice. On top of its conceptual elegance, a permutation has the advantages that it does not have a key schedule and that its inverse does not need to be implemented or efficient [xlirii].

The sponge construction is a simple iterated construction for building a function  $F$  with variable-length input and arbitrary output length based on a fixed-length permutation (or transformation)  $F$  operating on a fixed number  $b$  of bits. Here  $b$  is called the width [xlirii].

The sponge construction operates on a state of  $b = r + c$  bits. The value  $r$  is called the bitrate and the value  $c$  the capacity.



**Figure 8:** Sponge construction [xxix]

First, the input string is padded with a reversible **padding** rule and cut into blocks of  $r$  bits. Then the  $b$  bits of the state are initialized to zero and the sponge construction proceeds in two phases:

- In the absorbing phase, the  $r$ -bit input blocks are XORed into the first  $r$  bits of the state, interleaved with applications of the function  $f$ . When all input blocks are processed, the sponge construction switches to the squeezing phase.
- In the squeezing phase, the first  $r$  bits of the state are returned as output blocks, interleaved with applications of the function  $f$ . The user chooses the number of output blocks at will.

The last  $c$  bits of the state are never directly affected by the input blocks and are never output during the squeezing phase.

### 2.5.3.1. Hermetic sponge construction

Any attack against a sponge function implies that the permutation it uses can be distinguished from a typical randomly chosen permutation. This leads to this design strategy called *hermetic sponge*. [xxix]

Adopting the sponge construction and building an underlying permutation  $f$  that should not have any properties exploitable in attacks. These properties are called structural distinguishers by the KECCAK developers' team. [xxix]

In this approach, one designs a permutation  $f$  on  $b = r + c$  bits and uses it in the sponge construction to build the sponge function  $F$ .

In the hermetic sponge strategy, the capacity determines the claimed level of security, and one can trade claimed security for speed by increasing the capacity  $c$  and decreasing the bitrate  $r$

accordingly, or vice-versa. When a padding rule is used with particular properties, one can securely instantiate sponge functions with different rates with the same fixed-length permutation. The simplest padding rule satisfying these properties are called the **multi-rate padding**: it appends a single 1-bit, then a variable number of zeroes and finally another 1-bit.

#### 2.5.4. Bit padding

Bit padding can be applied to messages of any size.

A single set ('1') bit is added to the message and then as many reset ('0') bits as required (possibly none) are added. The number of reset ('0') bits added will depend on the block boundary to which the message needs to be extended. In bit terms this is "1000 ... 0000".

This method can be used to pad messages, which are any number of bits long. For example, a message of 23 bits that is padded with 9 bits in order to fill a 32-bit block:

```
... | 1011 1001 1101 0100 0010 0111 0000 0000 |
```

This padding is the first step of a two-step padding scheme used in many hash functions including MD5 and **SHA**.

## 2.6. Elliptic Curve Cryptography introduction

In Secret Key Cryptography (Block ciphers, Stream ciphers), a single key is used for both encryption and decryption cases. Because a single key is used for both functions, Secret Key Cryptography is also called Symmetric Encryption. For symmetric encryption to work, two nodes share the same secret key, which has to be protected from access by others. But, the process for installation of the key in the system is an important issue to solve by using only symmetric key. The main challenge is in the case of wide distributed area, where frequent key changes are required in unprotected areas, as the chance of an attacker learning the key is high. That is why, the cryptography system's strength rests on the key distribution technique.

Cryptosystems based on Elliptic Curve Cryptography are especially known for more efficient resource utilization than any other public key techniques. Also the bilinear pairings on elliptic curves can be used to create useful cryptosystems. Elliptic Curve Cryptography is used in many different commercial products such as mobile phones, smart cards, email systems and many others. So at some point it is logical to try to expand it in IoT devices. [iii] [vi]

In every cryptographic scheme the fundamental security lies in the hardness of the underlying mathematical problem. As hardness of mathematical problem increases it will provide more security as it is hard to hack. The difficulty of these problems directly impacts the performance, and also to the size of key parameters. This complexity of mathematical problem is inversely proportional to the processor time and battery consumption.

The hardness of Elliptic Curve Cryptography is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP). To find the hardness of ECC first scalar point multiplication should be known, and it is given by  $Q=sP$ , where  $Q$  and  $P$  are two points on a certain elliptic curve. The Elliptic Curve Discrete Logarithm problem occurs when the coordinates of  $P$  and  $Q$  are known and the scalar  $s$  needs to be calculated. This computational problem becomes harder with the size of domain parameters. So the most important aspect of this technology is the key size (as we will talk further in the next chapter) [xii]

The advantages that can be gained from smaller keys include not only faster computations and smaller memory requirements, but also energy savings for sensor devices because fewer bits are required to be transmitted.

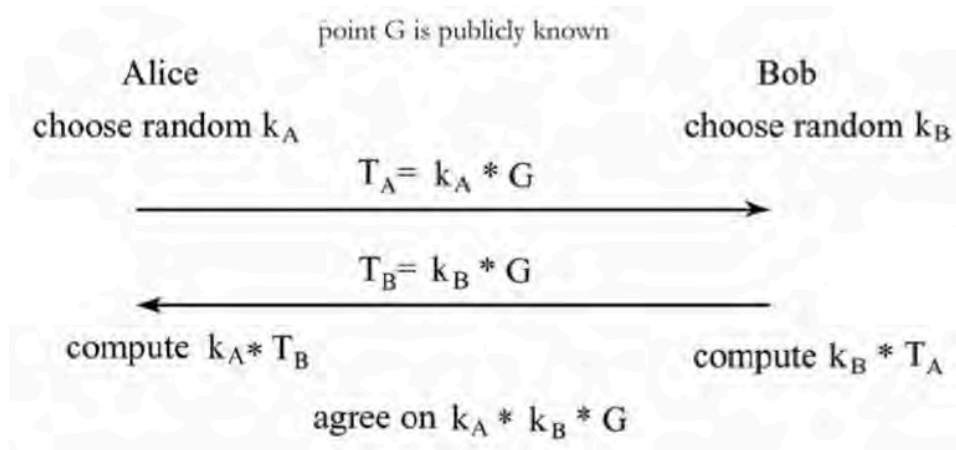
Generally speaking, all this advantages have made Elliptic Curve Cryptography the most attractive family of **public key algorithms** by researchers, especially in the case of wireless sensor networks. This is because although the key sizes are smaller (smaller than other public-key systems such as RSA) the same level of security is achieved leading to performance advantages that can be applied in these systems. [iii] [vi]

### 2.6.1. Mathematical resume – Alice and Bob example

The Alice and Bob example

Alice and Bob want to exchange a key. They carefully chose an elliptic curve  $E$  and a public base point  $G(x,y)$  on the curve

- Alice chooses her private key, a random integer  $k_A$  and Bob chooses a random integer  $k_B$ . The random integers are kept private
- Alice computes her public key, a new point on the elliptic curve by performing scalar multiplication  $T_A = k_A G$  and sends it to Bob who simultaneously computes his public key  $T_B = k_B G$



**Figure 9:** ECC Key computation [vi]

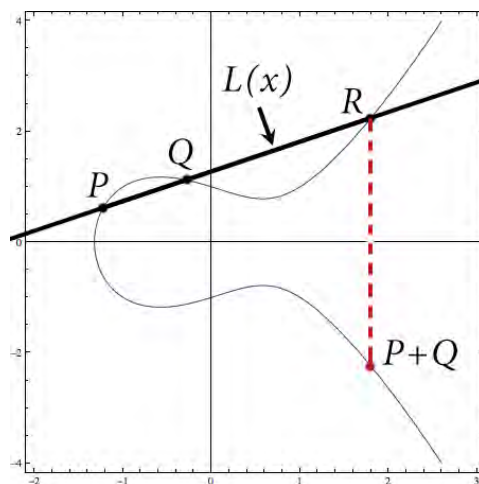
- Alice receives  $T_B$  and computes the shared secret, a new point on elliptic curve  $K = k_A T_B = k_A k_B G$ . Similarly, Bob takes  $T_A$  and computes  $K = k_B T_A = k_B k_A G$

### 2.6.2. Elliptic Curve definition

Just to put the problem in context. It is important to remember that an elliptic curve  $E$  is defined as the set of solutions  $(x,y) \in \mathbb{Z}_p \times \mathbb{Z}_p$  that satisfy the equation

$y^2 \equiv x^3 + ax + b \pmod{p}$  along with the point at infinity  $O$

- $a, b \in \mathbb{Z}_p$  are constants such that  $4a^3 + 27b^2 \neq 0 \pmod{p}$  and  $p > 3$
- The set of points on the curve with coordinates in a finite field along with the point of infinity  $O$  form groups with respect to addition operation
- $P + O = O + P = P$  for all  $P \in E$
- $P + Q = Q + P$  and  $(P + Q) + R = P + (Q + R)$  where  $P, Q, R \in E$



**Figure 10:** Elliptic Curve representation [vi]



### 2.6.3. Elliptic Curve Cryptography in IoT

Different studies talk about the viability of ECC as an effective tool to use in IoT. Many of these studies, where they use their own implementation of multiplication of points on elliptic curves, it is argued that PKI for secret keys' distribution is, in fact, tractable as well. These studies might not specifically focus on the trade-offs one should need to make in order to integrate PKI (Public Key Infrastructure) with particular applications but, rather, on whether PKI is viable at all. As it stated in "*Implementing Public Key Infrastructure for sensor networks*" by David J. Malan, Matt Welsh and Michael D. Smith, with elliptic curves over  $\mathcal{F}_{2^p}$ , generation of public keys requires no more than 34 seconds, and distribution of shared secrets requires no more than that, using just over 1 KB of SRAM and 34 KB of ROM. [xvii]

Communication costs, meanwhile, are minimal, with only 2 packets required for transmission of a public key among nodes. To be sure, not all sensor networks (nor their applications) require PKI, let alone any form of security. But with PKI comes capabilities that can certainly prove useful, among them the abilities to distribute symmetric keys securely and to sign messages digitally.

Other researches present the implementation of elliptic curve cryptography in the *MICAz Mote*, a popular sensor platform. Overall, from an initial state of the art point of view, it is fair to say that this type of Cryptography is not widely implemented in IoT sensors. But, as stated in the first paragraph, various studies examine ECC as an option. We will also go through that in the next chapters.



### 3. Methodology / Project Development

In order to carry out the thesis, data from previous studies was taken into account. Also, MATLAB simulations helped to understand some key processes like the hash function performance. After fulfilling all the documentation needs, supported by our simulations, a fair comparison between the different schemas (under well-defined environments) was performed, and we could draw the conclusions that will be presented in the fifth stage.

As it will be seen in this chapter, this thesis focuses on the gathering of previous work developed by different top tier professors, enhances the process of understanding it at a high level by someone whose knowledge of the topic is significantly inferior, and complements all of this with specific tests (MATLAB simulations) following the objective of trying to learn which cryptographic schema seems the most appropriated (focusing on performance parameters) for IoT devices. Also, in order to perform a better comparison we will take a standard for each case.

Regarding Symmetric Key Cryptography, as a Block Cipher we will use the current CLEFIA algorithm designed by Sony. As a Stream Cipher we will see the MICKEY v2 standard, an algorithm developed by Steve Babbage and Matthew Dodd. After that, we will focus on the SHA-3 Hash function, Hash algorithms are the top of Symmetric Key Cryptography.

After the Symmetric Key Cryptography study we will perform the comparison with the only Public Key Cryptography standard that, nowadays, seems to fit into the IoT world requirements. This is the Elliptic Curve Cryptography schema, especially known for more efficient resource usage than any other public key techniques.

#### 3.1. Keys and Cryptography

As a starting case, research was done focusing into the modern field of cryptography, which is basically divided in two main units; **Symmetric (Private) Key Cryptography** and **Asymmetric (Public) Key Cryptography**.

Just to put us in context, as the development of digital computers and electronics helped in cryptographic analysis, it made possible much more complex ciphers. Basically, computers allowed for the encryption of any kind of data representable in any binary format, while classical ciphers only encrypted written language messages. Basically, in classical cryptography neither the key nor the algorithm was known, while in modern cryptography the algorithm is known remaining the keys the only issue. It is also important to notice that although its application dates since 1940s, extensive open academic research into cryptography is relatively recent, beginning in the mid 1970s.

*Why shall we start talking about Symmetric Key Cryptography?* Basically because this was the only kind of encryption publicly known until June 1976. As we have seen in the previous chapters, symmetric key ciphers are implemented as **block ciphers**, **stream ciphers**. A block cipher enciphers input in blocks of plaintext as opposed to individual characters, which is the input form used by a stream cipher. Unlike block and stream ciphers, which are invertible, hash functions produce an output (called hashed-out) that cannot be used to retrieve the original input data. That is why hash functions are used to verify the authenticity of data retrieved from an untrusted source or to add a layer of security. The key concept to start the understanding of the **Symmetric Key Cryptography** is the *cipher* concept. After that, the transition to the Hash function form will

be natural, although as stated in the previous section, the Hash standard under study in this thesis, SHA-3, uses **KECCAK**, which is built on a cryptographic sponge instead of a pure cipher structure.

Once we are placed, before comparing ciphers performance applied into the IoT world, it is interesting to observe the evolution from one to another.

So basically, as asserted before, in Symmetric Key Cryptography the encipher and decipher process is carried out with the same *secret key*. This situation is perfectly summarized in the following schema:

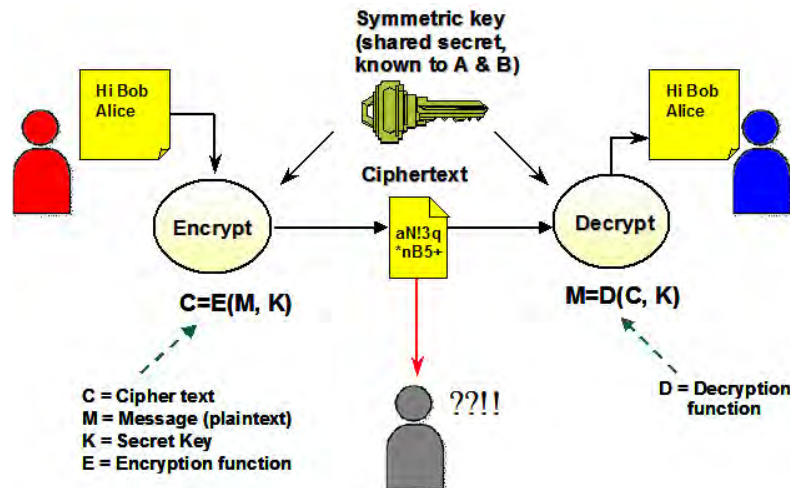


Figure 11: Symmetric Key Cryptography [xl]

As it can be perfectly seen, following this schema we can ensure both *confidentiality* and *authenticity*.

If we suppose to have a network of **N users** (in the IoT world that is N sensors, devices, etc.) that means that the total number of keys is  $\binom{N}{2} = \frac{N(N-1)}{2}$ . Also, using Private-Key cryptography, each user needs to store a total of N-1 keys.

If we take the Public Key Cryptography instead, each user possesses 2 keys, the private key and the public key. So, if a message is ciphered with one user's key, the complementary key is needed to decipher the Cryptogram. This is also perfectly described in the following schema:

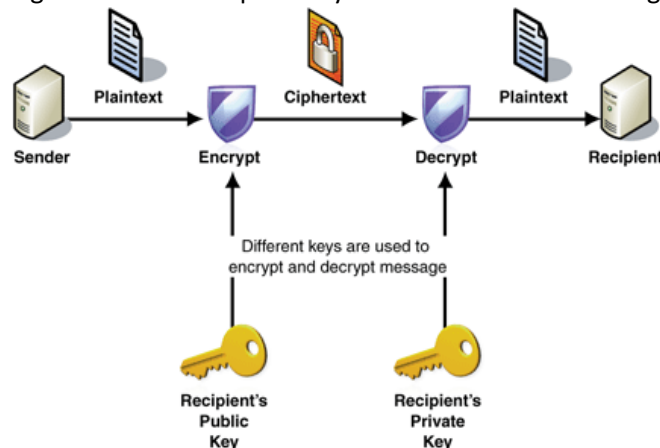


Figure 12: Public Key Cryptography [xli]

In that case, we can have three scenarios:

- **NO** confidentiality, but authenticity
- Confidentiality, but **NO** authenticity
- Both *confidentiality* and *authenticity*.

The first case is obtained by ciphering the message using the *Sender's* Secret Key.

The second case is obtained by ciphering the message using the *Recipient's* Public Key (as shown in figure 12).

The third case is obtained by sending the following Ciphertext:

$C = E_{K_{SA}}(E_{K_{PB}}(M))$  → That is enciphering the message with the *Recipient's* Public Key and then enciphering it again with the *Sender's* Secret Key. *\*Formula info at the bottom of the page.*

Thanks to this Public Key Cryptography schema we add the following property to the communication:

***“The authentication is now verifiable by a third part”***

Again, if we suppose to have a network of **N users** (N sensors, devices, etc.) that means that the total number of keys is  $2N$  (two keys per user existing in the system). Also, using Public Key cryptography, each user needs to store a total of  $N+1$  keys (the users own public and private key plus every other user public key).

So arrived at this point we see the first “dilemma”. *Is it really needed in an IoT network to perform an authentication validation by 3<sup>rd</sup> parties? Can we avoid this fact in order to make our devices faster, computationally speaking, by only using Symmetric Key Cryptography?*

In the following pages we will be looking at the performance studies that have been carried up until now in IoT environments, so we can obtain some important information that will help us to discern this questions. The truth, although, is that Cryptography (also in IoT) is not directly applied in the form of *Message-encryption-decryption* in the lonely form of a public or a private Key. Modern cryptography is implemented in the form of **Stream Ciphers** or **Block Ciphers** in the case of Symmetric Cryptography, and **Public Key Algorithms** (such as Elliptic Curve Cryptography) in the case of Public Key Cryptography.

*\*C → Cipher text*

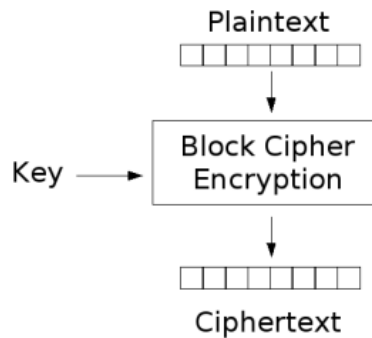
$E_{K_{SA}}$  → *Ciphering using Alice's (sender) Secret Key*

$E_{K_{PB}}$  → *Ciphering using Bob's (receiver) Public Key*

$M$  → *Message*

### 3.2. Symmetric Key Cryptography - The ciphers

Remembering; in a Stream cipher, each time an input symbol arrives an output ciphered symbol is generated. In a block cipher, the message is divided into blocks of  $K$  length, which are ciphered independently, implying that we have to use a device with memory.



**Figure 13:** Most basic Block Cipher schema

The vast majority of Symmetric Key algorithms are based on **Block Ciphers**, as in almost every modern device we have enough capabilities to perform the block ciphers required operations. The key aspects here are those environments where the cipher cost (performance cost) could be very relevant, such as the IoT world. It is precisely in those environments, where **Stream Ciphers** are useful, making the comparison between these two schemas is very relevant.

The first basic comparison between these two structures, only under theoretic aspects, is the following:

**Table 4:** Ciphers primitive comparison

| STREAM CIPHER                                    | BLOCK CIPHER   |
|--|--|
| Operates on smaller units of Plaintext           | Operates on larger blocks of data                            |
| Continuous process of input elements             | Block process of input at each time, producing output blocks |
| Less code required                               | More code required   |
| The same key is applied for every bit            | The key is applied at every block                            |
| Design more suitable for hardware implementation | Easier to implement on Software                              |

The ideal Stream Cipher objective would be to use a randomly generated key at every input, in the modern stream ciphers that is achieved by using pseudo-random sequences, following the Coulomb's random postulates. It is not an objective of this thesis to demonstrate how it is obtained, but to study the performance capabilities between different cipher schemas in an IoT environment. Let's start with the behaviour block ciphers schemas in an IoT environment.

### 3.2.1. CLEFIA

The first study taken into account is the one presented in: “*A Survey of Lightweight Cryptography Implementations*” by Thomas Eisenbarth (···) [ii], where the authors review recent developments in this area for symmetric and asymmetric ciphers, targeting embedded hardware and software. In the case of Symmetric Ciphers, which are really the points of interest in our case, the results will be presented next. We also take into account the following; the various ciphers used in that simulation were using 4bit S-boxes (Explained in 2.3.3). That was done this way because it can result in significant area savings in case serialized implementation is desired. Also the 4bit design was hardware driven, as 8bit S-boxes require, approximately, 40 times more area than 4 bit S-boxes. That is, precisely, an example of something that typically wants to be avoided in the IoT world, as devices are thought to be small. 4 bit S-boxes must be cryptographically weaker than 8 bit S-boxes, as they have less bits, which implies less computational capacity. Nevertheless, through careful selection, it is possible to obtain the appropriate security level. We assume that we achieve enough security level with 4 bit S-boxes (It has to be remembered that this thesis is focused on cryptography algorithms/schemas performance, assuming a correct level of security).

Resource efficiency (which will be mainly measured by memory consumption) is more critical than throughput, especially because many embedded applications encrypt only small payloads. The small microcontrollers used in those studies offer as little as tens of kilobytes of program memory, and sometimes less than 1 Kbyte of SRAM, plus they usually operate at clock speeds of few MHz. In this specific case, all the discussed ciphers were implemented for 8bit AVR microcontrollers (AVRs are a popular family of 8bit RISC microcontrollers). Also, in order to keep the source code small, a straightforward approach for software implementations was used, as only the substitution tables were designed as lookup tables (LUTs) [ii]. The LUTs are stored in the ROM memory of the microcontroller. Many “fast software” implementations of ciphers use larger LUTs because they then achieve a higher throughput, leading to a typically unacceptable increase in code size for the majority of embedded applications.

But the fact that these designs were full of limitation is not all bad news. For battery-powered devices, low computational complexity can be of great value, as processing time correlates directly with power consumption, a key aspect in those device designs.

| Cipher  | Key bits | Block bits | Cycles per block | Throughput at 100 kHz (Kbps) | Logic process      | Area (GEs) |
|---|----------|------------|------------------|------------------------------|--------------------|------------|
| Block ciphers   |          |            |                  |                              |                    |            |
| Present   | 80       | 64         | 32               | 200.00                       | 0.18 $\mu\text{m}$ | 1,570      |
| AES   | 128      | 128        | 1,032            | 12.40                        | 0.35 $\mu\text{m}$ | 3,400      |
| Hight   | 128      | 64         | 34               | 188.20                       | 0.25 $\mu\text{m}$ | 3,048      |
| Clelia  | 128      | 128        | 36               | 355.56                       | 0.09 $\mu\text{m}$ | 4,993      |
| mCrypton  | 96       | 64         | 13               | 492.30                       | 0.13 $\mu\text{m}$ | 2,681      |
| DES   | 56       | 64         | 144              | 44.40                        | 0.18 $\mu\text{m}$ | 2,309      |
| DESXL   | 184      | 64         | 144              | 44.40                        | 0.18 $\mu\text{m}$ | 2,168      |
| Stream ciphers  |          |            |                  |                              |                    |            |
| Trivium <sup>5</sup>  | 80       | 1          | 1                | 100.00                       | 0.13 $\mu\text{m}$ | 2,599      |
| Grain <sup>5</sup>  | 80       | 1          | 1                | 100.00                       | 0.13 $\mu\text{m}$ | 1,294      |
| *AES: Advanced Encryption Standard; DES: Data Encryption Standard; DESXL: lightweight DES with key whitening. |          |            |                  |                              |                    |            |

Figure 14: Lightweight Cryptography ciphers comparison [ii]

As it was said before, we will focus on the CLEFIA block cipher and the MICKEYv2 stream cipher. In those simulations we can check CLEFIA's performance:

**Table 5:** CLEFIA Data [ii]

|        | Key bits | Block bits | Cycles per block | Throughput at 100kHz (kbps) | Logic process | Area (GEs) |
|--------|----------|------------|------------------|-----------------------------|---------------|------------|
| CLEFIA | 128      | 128        | 36               | 355.56                      | 0.09 $\mu$ m  | 4,993      |

\*Cycles per block: Cycles are CPU instruction cycles. Cycles per byte roughly measure how many instructions, in a given instruction set, are needed to produce each byte of output. So in the case of a Block Cipher, like CLEFIAs case, it makes sense to know how many instructions are needed to produce each block of output. They're a reasonably good relative measure of the performance of different algorithms.

\*GE stands for **Gates Equivalents**, a unit of measure, which allows specifying manufacturing-technology-independent complexity of digital electronic circuits. It basically allows you to define the complexity of the production technology, regardless of the complexity of digital electronic circuits.

As it can be seen, this CLEFIA implementation uses both a 128 bit Key and Block size. That means a performance of 1:1 in the plaintext block encryption, as the key is as big as the plaintext block. As it can be also seen in **Figure 14** it is the best performance amongst the other block ciphers together with the AES, which also uses a 128 bit Key and Block size. The important difference here is the fact that, in the CLEFIA case, the cycles per block is 36, whereas in the AES algorithm we are talking about 1,032 cycles per block. That means a better performance of the CLEFIA algorithm, which needs significantly less instructions to produce each block of output. It also assumes a notable kbps throughput at 100kHz. Finally, regarding the area (GEs), the CLEFIA algorithm is the one presenting a bigger GEs area among all the compared block Ciphers of this study.

It was also interesting to compare CLEFIA performance with the other block cipher proposed and under consideration in **ISO/IEC 29192-2**

ISO/IEC 29192 defines security, classification and implementation requirements. 80 bit security is considered as the minimum security strength for lightweight cryptography and Lightweight cryptography is classified by a combination of the constraints on chip area, energy consumption, program code size, RAM size, communication bandwidth and execution time.

In "*Lightweight Cryptography for the Internet of Things*" by Masanobu Katagi and Shiho Moriai some comparisons are carried out, being the Hardware Properties of Lightweight Block Ciphers (directly related with the algorithm performance) the most interesting.



In the following figure, the gate efficiency comparison between the two algorithms is performed.

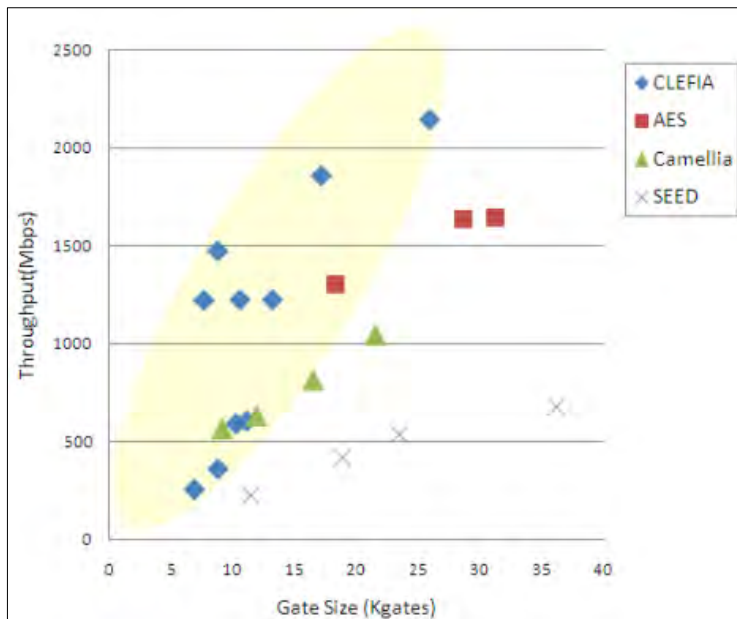


Figure 15: Gate efficiency comparison [xix]

Efficiency is defined as the ratio of Throughput (speed) to gate size (area). In this figure, a higher slope (in this case remarked with yellow area) indicates higher efficiency. The higher the efficiency is, the lower the energy consumption. This figure compares the CLEFIA block Cipher with three other conventional block ciphers like: AES (FIPS197 – USA’s Federal Information Processing Standard), Camellia (RFC3713 – Sony’s description of the Camellia Encryption Algorithm), and SEED (RFC4269 - The SEED Encryption Algorithm - IETF Tools). These ciphers are also used in TLS/IPsec. As it can be easily see CLEFIA has an advantage in gate efficiency over these ciphers, which is a key aspect when working on the IoT world.

If we decide to directly compare CLEFIA with the other algorithm under consideration in **ISO/IEC 29192-2, PRESENT**, the result shown in the next figure illustrates a comparison between the gate efficiency of these two ciphers on an application-specific integrated circuit (ASIC), an integrated circuit customized for a particular use, rather than intended for general-purpose use.

|  | mode    | block size<br>[bits] | key size<br>[bits] | cycle | area<br>[GE] | frequency<br>[MHz] | throughput<br>[Mbps] | technology<br>[ $\mu\text{m}$ ] |
|--|---------|----------------------|--------------------|-------|--------------|--------------------|----------------------|---------------------------------|
| Serialized Implementation (Area Optimization)        |         |                      |                    |       |              |                    |                      |                                 |
| PRESENT [6]  | enc     | 64                   | 80                 | 547   | 1075         | 0.1                | 0.0117               | 0.18                            |
| PRESENT [6]  | enc     | 64                   | 128                | 559   | 1391         | 0.1                | 0.0115               | 0.18                            |
| CLEFIA [1]   | enc     | 128                  | 128                | 176   | 2893         | 67                 | 49                   | 0.13                            |
| CLEFIA [1]   | enc/dec | 128                  | 128                | 176   | 2996         | 61                 | 44                   | 0.13                            |
| AES [5]  | enc     | 128                  | 128                | 177   | 3100         | 152                | 110                  | 0.13                            |
| AES [4]  | enc/dec | 128                  | 128                | 1032  | 3400         | 80                 | 10                   | 0.35                            |
| Round-based Implementation (Efficiency Optimization) |         |                      |                    |       |              |                    |                      |                                 |
| PRESENT [6]  | enc     | 64                   | 80                 | 32    | 1570         | 0.1                | 0.20                 | 0.18                            |
| PRESENT [6]  | enc     | 64                   | 128                | 32    | 1884         | 0.1                | 0.20                 | 0.18                            |
| CLEFIA [8]   | enc/dec | 128                  | 128                | 36    | 4950         | 201.3              | 715.69               | 0.09                            |
| CLEFIA [8]   | enc/dec | 128                  | 128                | 18    | 5979         | 225.8              | 1605.94              | 0.09                            |
| AES [7]  | enc/dec | 128                  | 128                | 11    | 12454        | 145.4              | 1691.35              | 0.13                            |
| AES [7]  | enc/dec | 128                  | 128                | 54    | 5398         | 131.2              | 311.09               | 0.13                            |

**Figure 16:** Hardware performance: CLEFIA vs PRESENT [xix]

\*Area (GE): As told in Table 5, it is a metric for cost and power consumption when the chip is clocked at a low frequency of a few hundred kHz, allowing the definition of the the complexity of the production technology.

\*Other Area definition: Amount of silicon used for the core design (excluding power rings and I/O cells). This result is typically expressed in  $\mu\text{m}^2$  for a specified process. However, the more usable process independent method of expressing the area is to calculate the Gate Equivalence (GE) of the total area by dividing by the lowest power two-input NAND gate's area

\*Frequency: Is the clock rate selected by the designer and applied as a constraint to the design tools. Those tools will make decisions to meet this requirement. Meaning that the higher the constraint is, the more area will be consumed.

\* The product of Area [GE] and Cycle is a metric for energy consumption.

This figure is showing how both ciphers can be implemented depending on the objective. This case shows both an Area Optimization implementation and an Efficiency Optimization implementation.

As shown in this figure, PRESENT uses 64 bit blocks while CLEFIA and AES are using 128 bit blocks. Typically, a 64 bit block cipher can be implemented with smaller gate counts, which is an obvious deduction as it uses less bits, this implementation takes certain security limitations but this thesis takes the assumption that the minimum security needs are accomplished in those studies.

Apart from that, it can also be seen how **CLEFIA implementations uses significantly less cycles to produce each block of output which indicates a higher overall efficiency** over PRESENT. The transistor technology [ $\mu\text{m}$ ] is similar in both cases whereas the throughput of CLEFIA is also significantly higher in the case of the CLEFIA block cipher, talking about difference orders between 3k6 and 8k, which indicates a better CLEFIA overall behaviour regarding global efficiency.

Finally in the **energy consumption** chapter both ciphers have a similar energy consumption ratio in the Area Optimization implementation, **being CLEFIA the one who consumes less energy**. When the studied implementation is the Efficiency Optimization, PRESENT has a notably better numbers regarding energy consumption, as it clearly consumes less energy than CLEFIA cipher.

### 3.2.2. MICKEY

So, at this point, a question is raised. *Why should we consider Stream ciphers if we are obtaining good performance level with Block cipher algorithms?* Also, security is known to be stronger for Block Cipher cases. The importance here is the cipher computational cost, **as a Stream cipher can be adapted to more memory constrained devices than a Block Cipher**. That is why; even though these ciphers are not used in other devices, it makes total sense to study them in this context.

Most of the stream ciphers are based on LFSR (Linear Feedback Shift Registers), which allow them to generate sequences with statistic properties similar to a random source, plus **they are extremely simple devices**.

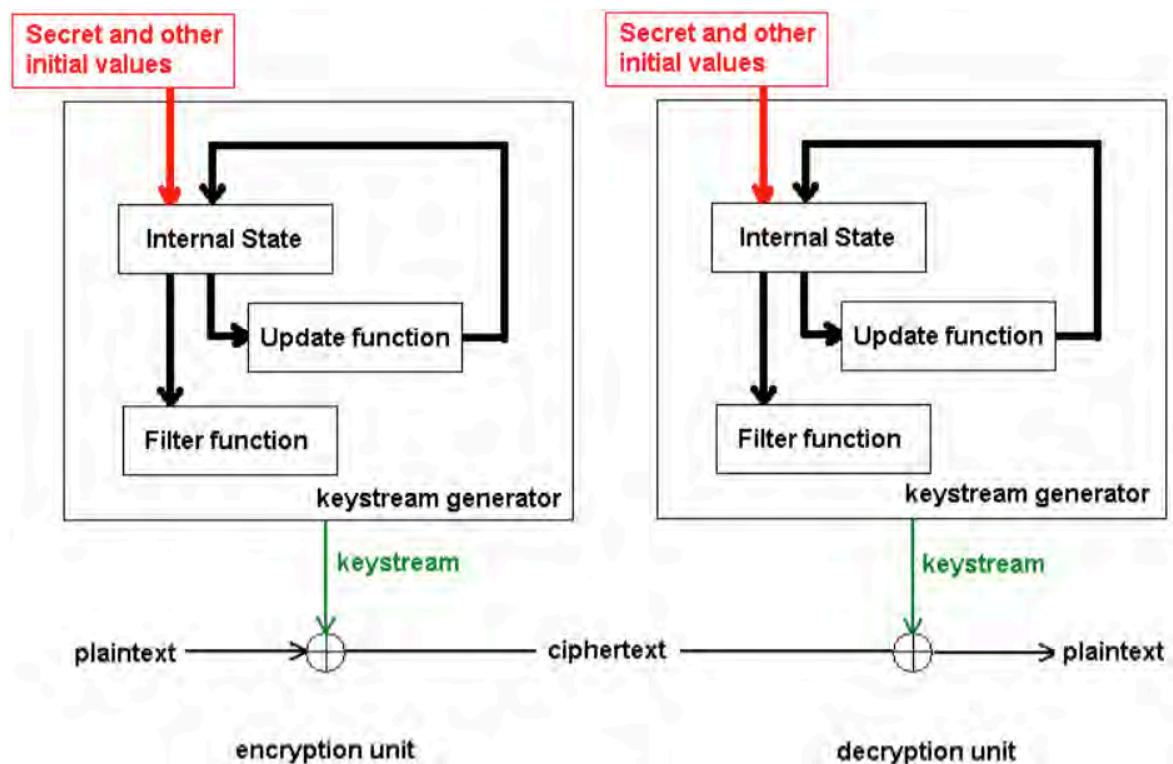


Figure 17: Stream Cipher scheme [xv]

The upper figure shows us a simplified schema of how a Stream Cipher operates. As it can be seen, the cipher itself is no more than a pseudo-random sequences generator. As it was stated earlier in this chapter, those sequences are the ones that shall fulfill Coulombs randomness postulates (which will not be studied or mathematically proven in this thesis):

- Equidistributed sequence
- Burst test
- Normalized autocorrelation

As stated in: “*A Survey of Lightweight Cryptography Implementations*” by Thomas Eisenbarth (...)  
[ii], stream ciphers had received little attention from the scientific community, but this started to

change with the increasing interest in stream ciphers in projects such as *eStream*, within the European Network of Excellence in Cryptography, which aimed to foster knowledge about stream ciphers. One of the studied stream ciphers was MICKEY 2.0 submitted by Steve Babbage and Matthew Dodd. From the *eStream* European project we can access the data form implementation studies of MICKEY v2. One of those studies is “*On the Hardware Implementation of the MICKEY-128 Stream Cipher*” by Paris Kitsos. It is stated how MICKEY v2.0 has two major advantages versus other stream Ciphers; The low hardware complexity, which results in small area and the high level of security.

As the other studies presented before, an FPGA device with low resources was used for the performance demonstration. The performance comparisons between the MICKEY v2 proposed system and previous published architectures of Stream Cipher will be shown in the next figure 18. As good comparison colleagues other well-known algorithms were chosen, such as A5/1 cipher (used in GSM mobile phones), W7 stream cipher (a cipher optimized for efficient hardware implementation at very high data rates), or the E0 algorithm that Bluetooth system uses.

| Stream Cipher | FPGA Device         | F (MHz) | Throughput (Mbps) |
|---------------|---------------------|---------|-------------------|
| A5/1 [4]      | 2V250FG25           | 188.3   | 188.3             |
| W7 [4]        | 2V250FG25           | 96      | 768               |
| RC4 [4]       | 2V250FG25           | 60.8    | 121               |
| E0 [5]        | 2V250FG25           | 189     | 189               |
| WG [6]        | ASIC                | 1000    | 125               |
| AES [8]       | Spartan II XC2S30-6 | 60      | 69                |
| AES [9]       | Spartan-II XC2S15-6 | 67      | 2.2               |
| Proposed      | XCV50ECS14          | 170     | 170               |

**Figure 18:** Stream Cipher HW performance comparison [xx]

As it can be seen in the above table, the proposed MICKEY cipher implementation (denoted with *Proposed*) while has a way better performance than other AES algorithm stream ciphers, also requires more hardware resources than those mentioned AES implementations. As it is also table illustrated, the proposed cipher implementation achieves a competitive clock frequency while showing an overall nice Mbps throughput performance if compared with the others Stream ciphers. Once again, the key factor of this design is the low level of FPGA utilization that it achieves, while it still competes with other ciphers that don't have this low FPGA usage. Going through complimentary hardware efficiency and using this synthesis results it is proven that this design is more than suitable for area restricted hardware implementations.

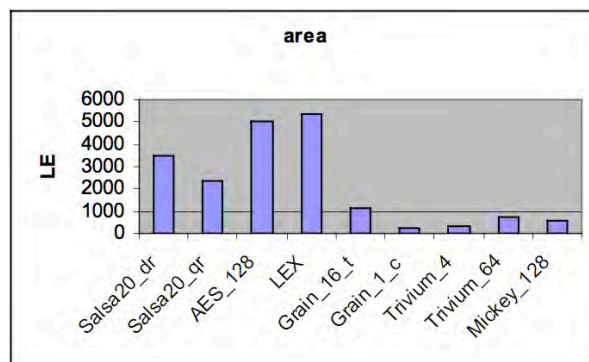
It is also important to highlight the fact that the author believes that MICKEY-128 cipher can still be implemented in a more compact manner, for example, using only one register. That would make this cipher still more suitable for IoT devices as compactness goes in favour of the technology application in IoT networks. This research (which is not on the scope of this work) is believed to be a good task for other developers to implement.

Continuing the research on the MICKEY v2 stream cipher, following the eStream project work, various eStream candidates were evaluated at a hardware level using ALTERA FPGAs (Intel FPGAs

that offer a wide variety of configurable embedded, SRAM etc.). In “*Hardware evaluation of eSTREAM Candidates*” by Marcin Rogawski [xv], the comparison between different algorithms is performed under the following 6 assumptions: Only standard logic cells were used (1), standardized interface (2), the algorithms were implemented without any authentication method add-on (3), the code was written in VHDL language (4), Quartus II 6.0 and Cyclone family (5) and Cyclone\_power\_est\_2-12 was used for power consumption’s estimation (6).

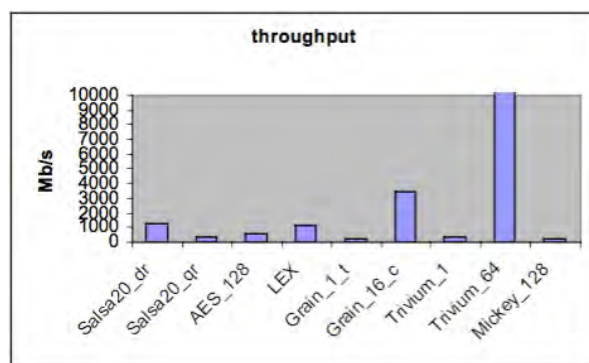
During the tests that were run in this project, the following was observed from MICKEY; It was defined as a compact algorithm, very simple to implement whose reference C-code and documentation very easy to follow [*attached in the annex section*]. The weakest point of this design was defined as its difficulty to be implemented with parallel realization, which was on of the designs proposed by the investigators.

After running the different ciphers implementation the observed results where the following:



**Figure 19:** Compactness summary [xv]

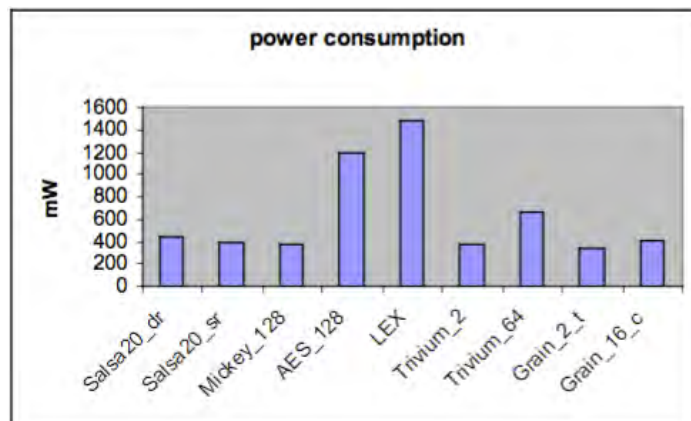
The figure 19 shows us that the MICKEY algorithm is on the top 3 in terms of compactness. *What does it mean?* It is basically proving us that **this stream cipher is easily implementable in small devices**, as less than 1000 LEs (\*) (537 LE, indeed) are only needed. We can also see how is the only 128 bit key algorithm ranked among the most compact (as it is based on Shift Registers).



**Figure 20:** Throughput summary [xv]

(\*) LE: Area comparison measuring unit (taking Altera LE microcontroller as reference)





**Figure 21:** Power consumption summary [xv]

These two upper figures (20 and 21) stress the idea that MICKEY v2 is a suitable candidate for IoT networks, but this fact has its own costs. As figure 21 shows, the power consumption of the device running the cipher algorithm is top 2 in terms of lower energy consumption, probably **one of the top performance features** desired on the Internet of Things world. This fact, though, has one major setback, which is the lower throughput ratio, being the worst of these tests staying at 220Mb/s. Deriving from this outcome, a question will be discussed in the *Results and Conclusions* sections, *is this Throughput enough for or not for IoT environments?* Is throughput, in fact, a *key feature in an IoT network?* Apart from that, the fact that it has the worst throughput ratio cannot be eluded.

One final eStream report was taken into account in our stream cipher study, “*Hardware results for selected stream cipher candidates*” by T. Good and M. Benaissa aims the following idea: Performance results are vital points to focussing the security analysis efforts on the **low resource cipher candidates**, that is done in order to provide an independent set of HW testing results for the algorithm candidates to expand the understanding of the various performance merits is also a key aspect to take into account.

It is also important to remember that this study was carried out under the following premise; Firstly, only candidates which are “free- for-all” were considered and secondly, only those ciphers that were seen from a low resource hardware perspective (candidates that were in the eStream project phase 2). Also, as the authors had no affiliation with any of the “ciphers developer teams”, the review was carried out independently of the eStream project.

It is really interesting to see how this study cuts various candidates under cryptanalysis standards first (obviously due to security concerns) and under area standards secondly, following on of the eStream project premises (which we didn’t mention before), “*candidates should be smaller than the AES*”. That is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST), which is actually a block Cipher. This premise makes totally sense, as we are studying stream ciphers, precisely, because they are generally a lesser-cost implementation of ciphers, what we pursue in IoT networks.

This study also addressed two new ideas, which will also be brought up on the Conclusions chapter: The first one is the fact that the comparison of designs is performed without taking into account Message Authentication Code (MAC) support (this is also something out of the scope of our thesis). Secondly, performance in terms of throughput only, will not address latency issues, something that may be critical in some IoT applications.

RFID applications and IoT applications place limits on power, area and latency directly. Excesses in any of these aspects would make a candidate unsuitable for the application. RFID tags have to be fundamentally low cost, implying low area. The exact same concept is applied to IoT devices. That raises the following conclusion: **A good metric for performance would be power-latency product versus area.**

The following test results are presented:

| Design        | Clock Frequency, MHz | Estimated Power, uW | Energy/bit, pJ/bit | Area-Time, um <sup>2</sup> -us | Tput/Area, kbps/um <sup>2</sup> | Power-Area-Time nJ-um <sup>2</sup> |
|---------------|----------------------|---------------------|--------------------|--------------------------------|---------------------------------|------------------------------------|
| Grain80       | 10.000               | 109.4               | 10.95              | 671                            | 1.490                           | 73.4                               |
| Grain80, x4   | 2.500                | 34.1                | 3.41               | 870                            | 1.150                           | 29.6                               |
| Grain80, x8   | 1.250                | 22.9                | 2.29               | 1136                           | 0.880                           | 26.0                               |
| Grain80, x16  | 0.625                | 19.5                | 1.95               | 1679                           | 0.596                           | 32.7                               |
| Trivium       | 10.000               | 181.2               | 18.12              | 1347                           | 0.742                           | 244.1                              |
| Trivium, x4   | 2.500                | 49.2                | 4.92               | 1379                           | 0.725                           | 67.9                               |
| Trivium, x8   | 1.250                | 28.8                | 2.88               | 1452                           | 0.689                           | 41.9                               |
| Trivium, x16  | 0.625                | 19.9                | 1.99               | 1651                           | 0.606                           | 32.9                               |
| Trivium, x32  | 0.313                | 16.1                | 1.61               | 1963                           | 0.509                           | 31.6                               |
| Trivium, x64  | 0.156                | 16.4                | 1.64               | 2551                           | 0.392                           | 41.7                               |
| F-FCSR-H      | 1.250                | 40.6                | 4.06               | 2468                           | 0.405                           | 100.3                              |
| Grain128      | 10.000               | 167.7               | 16.77              | 962                            | 1.039                           | 161.4                              |
| Grain128, x4  | 2.500                | 48.7                | 4.87               | 1104                           | 0.906                           | 53.7                               |
| Grain128, x8  | 1.250                | 29.9                | 2.99               | 1290                           | 0.775                           | 38.6                               |
| Grain128, x16 | 0.625                | 22.4                | 2.24               | 1653                           | 0.605                           | 37.0                               |
| Grain128, x32 | 0.313                | 21.9                | 2.19               | 2394                           | 0.418                           | 52.3                               |
| Mickey128     | 10.000               | 310.7               | 31.07              | 2612                           | 0.383                           | 811.6                              |
| Phelix, ½ rnd | 0.625                | 80.5                | 8.05               | 6822                           | 0.147                           | 548.9                              |
| Phelix, 1 rnd | 0.313                | 71.5                | 7.15               | 7793                           | 0.128                           | 557.1                              |
| Sosemaunk     | 0.313                | 57.9                | 5.79               | 9756                           | 0.103                           | 564.8                              |
| Salsa20, 1h   | 10.059               | 712.5               | 71.25              | 6286                           | 0.159                           | 4479.0                             |
| Salsa20, 4h   | 1.895                | 185.6               | 18.56              | 6694                           | 0.149                           | 1242.3                             |
| Salsa20, 16h  | 0.723                | 200.356             | 20.04              | 8499                           | 0.118                           | 1702.8                             |
| Salsa20, 32h  | 0.527                | 211.208             | 21.12              | 9656                           | 0.104                           | 2039.4                             |
| AES [12]*     | 4.219                | -                   | -                  | 2798                           | 0.357                           | -                                  |
| AES [13]*     | 80.625               | -                   | -                  | 1763                           | 0.567                           | -                                  |
| Better is:    | lower                | lower               | lower              | lower                          | higher                          | lower                              |

**Figure 22:** Metrics for an output rate of 10 Mbps (estimated typical wireless LAN) [xvi]

| Design        | Throughput, Mbps | Estimated Power, uW | Energy/Bit, pJ/bit | Area-Time, um <sup>2</sup> -us | Tput/Area, kbps/um <sup>2</sup> | Power-Area-Time, nJ-um <sup>2</sup> | Latency, us | Power-Area-Latency, uJ-um <sup>2</sup> | Power-Latency, nJ |
|---------------|------------------|---------------------|--------------------|--------------------------------|---------------------------------|-------------------------------------|-------------|--|-------------------|
| Grain80       | 0.100            | 3.3                 | 33.0               | 67,098                         | 0.0149                          | 221.2                               | 3,210       | 71.0                                   | 10.58             |
| Grain80, x4   | 0.400            | 4.5                 | 11.2               | 21,747                         | 0.0460                          | 97.3                                | 810         | 31.5                                   | 3.63              |
| Grain80, x8   | 0.800            | 6.1                 | 7.6                | 14,198                         | 0.0704                          | 86.5                                | 410         | 28.4                                   | 2.59              |
| Grain80, x16  | 1.600            | 9.3                 | 5.8                | 10,493                         | 0.0953                          | 97.9                                | 210         | 32.9                                   | 1.96              |
| Trivium       | 0.100            | 5.6                 | 56.1               | 134,715                        | 0.0074                          | 755.7                               | 13,330      | 1007.3                                 | 74.77             |
| Trivium, x4   | 0.400            | 5.9                 | 14.6               | 34,469                         | 0.0290                          | 201.7                               | 3,360       | 271.1                                  | 19.67             |
| Trivium, x8   | 0.800            | 6.4                 | 8.0                | 18,153                         | 0.0551                          | 116.2                               | 1,700       | 158.0                                  | 10.88             |
| Trivium, x16  | 1.600            | 8.1                 | 5.1                | 10,318                         | 0.0969                          | 83.6                                | 870         | 116.3                                  | 7.05              |
| Trivium, x32  | 3.200            | 10.3                | 3.2                | 6,135                          | 0.1630                          | 62.9                                | 450         | 90.6                                   | 4.61              |
| Trivium, x64  | 6.400            | 14.3                | 2.2                | 3,986                          | 0.2509                          | 57.0                                | 240         | 87.6                                   | 3.43              |
| F-FCSR-H      | 0.800            | 10.6                | 13.2               | 30,847                         | 0.0324                          | 326.5                               | 2,250       | 587.8                                  | 23.82             |
| Grain128      | 0.100            | 4.3                 | 43.5               | 96,250                         | 0.0104                          | 418.5                               | 5,130       | 214.7                                  | 22.31             |
| Grain128, x4  | 0.400            | 5.6                 | 14.0               | 27,588                         | 0.0362                          | 154.5                               | 1,290       | 79.7                                   | 7.23              |
| Grain128, x8  | 0.800            | 6.9                 | 8.6                | 16,127                         | 0.0620                          | 111.3                               | 650         | 57.9                                   | 4.48              |
| Grain128, x16 | 1.600            | 9.3                 | 5.8                | 10,333                         | 0.0968                          | 96.7                                | 330         | 51.1                                   | 3.09              |
| Grain128, x32 | 3.200            | 14.8                | 4.6                | 7,480                          | 0.1337                          | 110.5                               | 170         | 60.1                                   | 2.51              |
| Mickey128     | 0.100            | 11.2                | 111.7              | 261,204                        | 0.0038                          | 2,917.6                             | 4,170       | 1216.6                                 | 46.58             |
| Phelix, ½ rnd | 1.600            | 32.9                | 20.6               | 42,635                         | 0.0235                          | 1,404.8                             | 510         | 1146.3                                 | 16.80             |
| Phelix, 1 rnd | 3.200            | 41.6                | 13.0               | 24,352                         | 0.0411                          | 1,014.1                             | 340         | 1103.4                                 | 14.16             |
| Sosemaunk     | 3.200            | 41.3                | 12.9               | 30,487                         | 0.0328                          | 1,260.3                             | 2,550       | 10284.0                                | 105.41            |
| Salsa20, 1h   | 0.099            | 26.3                | 264.0              | 632,312                        | 0.0016                          | 16,598.8                            | 5,330       | 8795.6                                 | 139.92            |
| Salsa20, 4h   | 0.528            | 31.0                | 58.7               | 126,828                        | 0.0079                          | 3,926.7                             | 1,000       | 2072.7                                 | 30.96             |
| Salsa20, 16h  | 1.384            | 54.8                | 39.6               | 61,416                         | 0.0163                          | 3,366.0                             | 400         | 1863.1                                 | 21.92             |
| Salsa20, 32h  | 1.896            | 68.5                | 36.1               | 50,920                         | 0.0196                          | 3,486.3                             | 300         | 1983.3                                 | 20.54             |
| AES [12]*     | 0.237            | -                   | -                  | 118,054                        | 0.0085                          | -                                   | 500         | -                                      | -                 |
| AES [13]*     | 0.001            | -                   | -                  | 1,421,064                      | 0.0007                          | -                                   | 10,160      | -                                      | -                 |
| Better is:    | higher           | lower               | lower              | lower                          | higher                          | lower                               | lower *     | lower                                  | lower ***         |

**Figure 23:** Metrics operating at 100kHz clock (low-end RFID/WSN applications) [xvi]

Those two figures, 22 and 23, present the two most interesting runs, which show the metrics under two IoT scenarios. Those scenarios are output rate of 10 Mbps (estimated typical wireless LAN) and operating at 100kHz clock (low-end RFID/WSN applications).

Focusing on the MICKEY cipher, in the 10Mbps output rate case we can see the following. This algorithm shows a decent Throughput/area metric, standing at  $0.383 \text{ kbps}/\mu\text{m}^2$ , proving the compactness of this cipher, taking into account that it does not present a good throughput metric. But what really surprises from those results is the power-area-time metric, resulting into a higher value than the one expected taking into account the other studies, where the **MICKEY algorithm stranded as a low consumption algorithm**.

Regarding the low-end RFID/WSN applications we see how, once more, the throughput rate results in its worst metric as it only reaches a total of 100kbps.

Nevertheless, as an 128 bit key stream cipher, according to their key size and the most relevant hardware performance metrics for the application area in a suggested priority order for further cryptanalysis effort, it can be stated that the MICKEY v2 algorithm is suitable for WSN applications although the best performance 128 key size stream cipher in this study is clearly the Grain128.



### 3.3. Hash function

After checking stream cipher algorithms performance we will now focus on the study of cryptographic hash functions, more precisely, into the Secure Hash Algorithm 3, **SHA-3**. As explained in chapter 2, *State of the art of the technology used or applied in this thesis*, after several cryptographic hash algorithms were successfully attacked around the years 2004-05 the NIST decided to held workshops in order to develop a new algorithm for standardization through a public competition. That new hash algorithm would be referred to as SHA-3. The winning algorithm was KECCAK, an algorithm that can also be used for authentication, authenticated encryption and pseudo-random number generation. With that being said, we should now focus on the properties that make it suitable for IoT networks.

The most famous public key cryptography algorithm is probably RSA (Rivest, Shamir, Adenan), which is based on the difficulty of both factoring large integers and working with the discrete logarithm. In this algorithm, both the public and the private key of a user (device) came from a large number obtained when multiplying two prime numbers ( $p$ ,  $q$ ) (not close prime numbers). It is proven [attached in the annex section] that this algorithm can provide confidentiality or authenticity. This algorithm, though, had a security scratch that would make an attacker able to supplant the sender's identity. That was because the attacker could find the message by using the sender's signature. One of the solutions to this problem was the Hash function.

The basic prerequisites for the hash function are the following:

- Input data could be any length
- Output data has a fixed length
- $H(x)$  is easy to calculate for any given  $x$  value
- $H(x)$  is unidirectional, that means that given any  $H(x)$  it is computationally impossible to obtain  $x$ .

\*Where  $H(x)$  is the hash function.

As we will see, Hash functions actually send digitally signed messages. *Is this applicable to the IoT world? If the answer is yes, is it viable with potential scenarios of hundreds of devices sending digitally signed information to each other?*

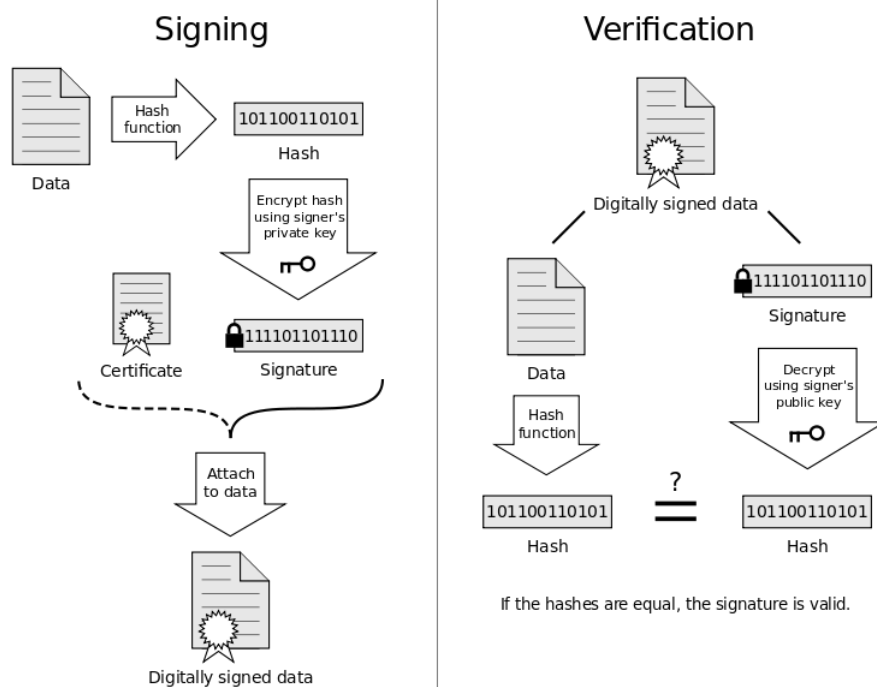


Figure 24: Hash function signing / verification process

Once the hash function context is set, let's continue with SHA-3 and KECCAK. As said in chapter 2, this algorithm uses a concept known as the **sponge construction**. It can also be seen as a **generalization of both hash functions**, which have a fixed output length **and stream ciphers**, which have a fixed input length. It operates on a finite state by iteratively applying the inner permutation to it, interleaved with the entry of input or the retrieval of output.

Some resumes state that SHA-3 is not suitable for IoT networks, in "Lightweight Cryptography for the Internet of Things" by Katagi [xix]. It is said that SHA-3 did not accomplish the necessary Lightweight cryptography properties, while some other papers actually implement SHA-3 libraries and test its potential usability for IoT networks.

On the study "Efficient and Concurrent Reliable Realization of the Secure Cryptographic SHA-3 Algorithm" by Siavash Bayat-Sarmadi [viii] a synthesis for SHA-3 algorithm is presented. An ASIC was chosen based on the resources available to the researchers in terms of library and tools. Another interesting point was that the presented schemes were not dependent on the hardware platform. Also, it is important to note that similar overheads are expected if FPGAs are used for the implementations of the schemas run in those simulations. In this paper, the highlight is both the synthesis of the original KECCAK algorithm and its error detection approach. Also, another approach called the RERO-based approach is presented and compared; this one is suitable for resource-constrained applications like IoT devices. The experiments use two restrictions: In the first one, a period constraint was imposed to achieve the working frequency of more than 650 MHz for the original implementation of KECCAK, allowing higher performance of the original structure but also costing a higher total area. In the second set of simulations a restriction was applied by loosening the period constraint, which resulted in lower working frequency for the original architecture, but also a lower area. Also, a variant of KECCAK, which operates one round in each cycle, was considered to derive the hashed output.

The results of this study are presented in the following figure:

| Algorithm            | Architecture                            | Area   |      | Frequency [MHz] | Throughput [Gbps]                               |
|----------------------|---|--|------|-----------------|---|
|                      |   | $[\mu m^2]$                                      | kGE  |                 |   |
| Keccak-f[1600] (512) | a. Original (with frequency constraint) | 66,306   | 47.0 | 676             | 28.8  |
|                      | b. Original (with loose constraint)     | 63,994   | 45.4 | 632             | 26.9  |
|                      | RERO-based scheme                       | 69,249<br>(w.r.t (a): 4.4%)<br>(w.r.t (b): 7.6%) | 49.1 | 1,192           | 25.4<br>(w.r.t (a): 11.8%)<br>(w.r.t (b): 5.9%) |

**Figure 25:** SHA-3 Performance results under authors ASIC Syntheses [viii]

As seen in the upper figure; the area overhead and throughput degradation for the RERO-based scheme for KECCAK is 4.4% and 11.8% in comparison with the original KECCAK scheme, respectively. On the other hand, in the case where these are loosened, the overheads are 7.6% and 5.9%, respectively.

In our thesis, this particular KECCAK implementation (RERO-based) is not important. The key idea behind all of this is that different implementations of the SHA-3 algorithm can be developed and compared to the original implementation. This one in particular, achieves worst performance results than the original architecture, while it follows a different error detection implementation. It is proven that KECCAK is simple enough to be studied and present different HW implementations of it, which is also something desirable in the IoT world. Apart from that, as it is proven by these same results, this technology, although being one order of magnitude superior in terms of area and kGE could still be applied to small devices. At the same time the throughput is way higher than all the Stream ciphers or Block ciphers that we have seen, directly implying that this is an algorithm to consider and take into account for some IoT applications needing high throughput levels.

These results also show how **Public Key Cryptography** concerns are real. The application of these kinds of algorithms is more expensive in terms of hardware than **Symmetric Key** algorithms that are lighter and, therefore, better suited for IoT networks.

### 3.3.1. SHA-3 Algorithm, Hash implementation simulation

As Hash functions are a more complex algorithm, it was interesting to implement/test its functionality via simulations in order to better understand how they actually work. Not only that, as a well-defined algorithm like SHA-3 was being studied, we wanted to assay its implementation via a stable environment like MATLAB.

To do so, firstly Dr. Markku-Juhani O. Saarinen function package [annex] was taken as a basis, as it builds an implementation intended for studying the algorithm, (not for a productive use). The code is designed to run on 64-bit little-endian platforms with gcc. The `main.c` module contains self-tests for all the hash sizes that are being supported. Basically it helps to simulate a whole run while having an outlook for the step by step. The function package is called `tiny_sha3`.

As commented above, this package was taken as a basis to study the KECCAK algorithm functioning at a high level (it is not intended to prove the sponge construction functioning in a mathematical point of view). After this revision, a “step-back” was performed when testing a general Hash function MATLAB implementation. This specific order was followed, because it was thought to be more interesting to test a specific hashing function once having firstly obtained the overall SHA-3 coded algorithm idea rather than implementing the sponge construction to prove that KECCAK algorithm works (something that has already been proven), which was never this thesis objective. A general hashing function is easier to understand and test so it was thought to be a more efficient way to learn and get introduced into this technology.

So let's take a look at the important phases of the hashing process implementation:

```
% Create the hash value: -----
if isFile
    % Open the file:
    FID = fopen(Data, 'r');
    if FID < 0
        % Check existence of file:
        Found = FileExist_L(Data);
        if Found
            Error_L('CantOpenFile', 'Cannot open file: %s.', Data);
        else
            Error_L('FileNotFound', 'File not found: %s.', Data);
        end
    end
end

% Read file in chunks to save memory and Java heap space:
Chunk = 1e6;      % Fastest for 1e6 on Win7/64, HDD
Count = Chunk;   % Dummy value to satisfy WHILE condition
while Count == Chunk
    [Data, Count] = fread(FID, Chunk, '*uint8');
    if Count ~= 0 % Avoid error for empty file
        Engine.update(Data);
    end
end
fclose(FID);

% Calculate the hash:
Hash = typecast(Engine.digest, 'uint8');
```

**Figure 26:** Hashing phase 1 - DataHash function

```

elseif isBin % Contents of an elementary array, type tested already:
    if isempty(Data) % Nothing to do, Engine.update fails for empty input!
        Hash = typecast(Engine.digest, 'uint8');
    else % Matlabs TYPECAST is less elegant:
        if isnumeric(Data)
            if isreal(Data)
                Engine.update(typecast(Data(:), 'uint8'));
            else
                Engine.update(typecast(real(Data(:)), 'uint8'));
                Engine.update(typecast(imag(Data(:)), 'uint8'));
            end
        elseif islogical(Data) % TYPECAST cannot handle LOGICAL
            Engine.update(typecast(uint8(Data(:)), 'uint8'));
        elseif ischar(Data) % TYPECAST cannot handle CHAR
            Engine.update(typecast(uint16(Data(:)), 'uint8'));
            % Bugfix: Line removed
        end
        Hash = typecast(Engine.digest, 'uint8');
    end
else % Array with type:
    Engine = CoreHash(Data, Engine);
    Hash = typecast(Engine.digest, 'uint8');
end

```

**Figure 27:** Hashing phase 2 - DataHash function

The hashing phase showed in the above two figures firstly check the input data type (`isFile` or `isBin`) which is previously defined before the hashing phase starts. After that the java engine used to perform the hashing is updated: `Engine.update(Data)` and the Hash is automatically calculated using `typecast` function. If the input data type is contents an array, the same Engine structure is updated by using function `CoreHash(Data, Engine)`.

In the following image we can see the above-mentioned initialization of the Engine structure, performed before the hashing:

```

% Create the engine: -----
try
    Engine = java.security.MessageDigest.getInstance(Method);
catch
    Error_L('BadInput2', 'Invalid algorithm: [%s].', Method);
end

```

**Figure 28:** Engine structure initialization

Once again, it is not this thesis objective to understand/demonstrate the usage of this method, in this case the **Engine** java method. It is just shown that this particular Hash function implementation takes advantage of it in order to perform the programming.

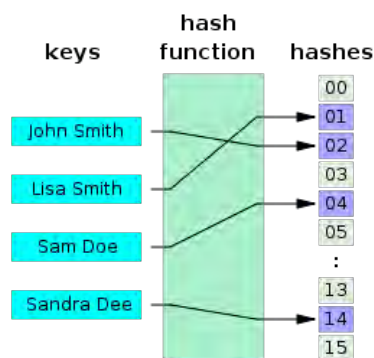
After that, the Hash is adapted to the specific output format that we have selected.

```
% Convert hash specific output format:
switch OutFormat
case 'hex'
    Hash = sprintf('%.2x', double(Hash));
case 'HEX'
    Hash = sprintf('%.2X', double(Hash));
case 'double'
    Hash = double(reshape(Hash, 1, []));
case 'uint8'
    Hash = reshape(Hash, 1, []);
case 'base64'
    Hash = fBase64_enc(double(Hash));
otherwise
    Error_L('BadOutFormat', ...
        '[Opt.Format] must be: HEX, hex, uint8, double, base64.');
```

**Figure 29:** Hash output format

So at this exact point we have the Hash message created. This hash is then encrypted using the sender's private key, creating the data signature. The message (data) would then be sent along with its own signature. So at the receiver's end all it has to do is Hash the received data with the same Hashing function and decrypt the sender's signature using the sender's public key. Once this process is finished all the receiver has to do is comparing the two results, if the hashed message equals the decrypted signature that means that the message has been correctly sent. No 3<sup>rd</sup> party intrusions or identity supplant have occurred.

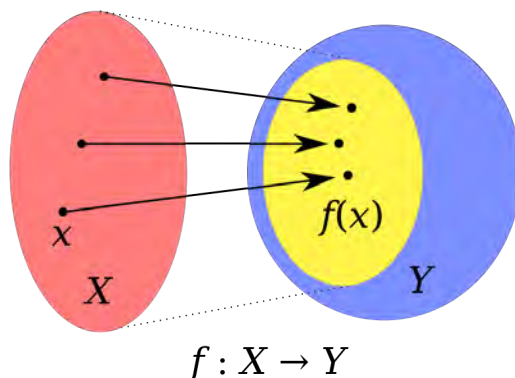
As you may be thinking, this moves away from the project focus that we have been seeing until now, which is basically **comparing cryptographic tools** solely from a **performance point of view**. This incise was produced in order to better understand this technology development. From a personal point of view, I felt that this Hash function concept was great and imaginative idea in order to avoid identity supplant, as an extra addition to actual cryptography. What's more, this application can perfectly be applied to the IoT networks. In some specific applications we may be interested to know certain information about devices, at the same time it may be crucial to determine whether this information is 100% veracious or not, in these application the fact that the information is encrypted or not, would go to a second term of importance.



**Figure 30:** Perfect Hash function example

As shown in the upper figure the ideally desired Hash function behaviour would be the perfect hash function. In this case, the hashed data would never collide, as the target set region is larger that the Hashing function domain. This is perfectly seen in the upper figure.





**Figure 31:** Mathematical display of a Hashing function  $f$

So, to remember, the KECCAK algorithm makes use of the **sponge construction**, a **generalization of both hash functions**, which have a fixed output length, as we have just seen in this simulation chapter **3.3.1**, and **stream ciphers**, which have a fixed input length as we have also seen in the previous chapters.

### 3.4. Elliptic Curve Cryptography - studies and IoT applications

As we commented during section 3, Public Key cryptography concerns are real; the implementation of these algorithms may be too much expensive for the typically tiny IoT devices. *Why is that there are still studies involving the usage of Public Key Cryptography?* The answer is Elliptic Curve Cryptography, a method developed in the mid 1980s with **some interesting properties that make it suitable for IoT networks**.

From now on, several studies have also been carried out, indeed there are actual implementations of ECC algorithms in specific sensor platforms (for example MICAz Mote). In this section we will explore different implementations and we will see its performance results so we can compare the behaviour of this technique with all the other studied protocols.

As assessed in “*Efficient Implementation of Elliptic Curve Cryptography in Wireless Sensors*” by Diego F. Aranha [x], Wireless sensor network features motivate the search for increasingly efficient algorithms and implementations of ECC for its devices. This work in particular shows that a specific implementation of Elliptic Curves Cryptography, called binary fields, offers significant computational advantages over prime curves when implemented in these Wireless Sensor Networks. The focus of this study lies on the performance side of this algorithm, assuming that the original simple one-pass Elliptic Curve Diffie-Hellman protocol is employed for key agreement. Under this premise, different ECC implementations are compared in terms of performance by the cost of multiplying integers by a random elliptic curve point. Also, it is shown how the ECC techniques have improved through the years, obtaining better results in terms of execution time for scalar multiplications of a random point of the elliptic Curve, the following figure shows these results on a MICAz Mote device:

| Finite field | Work           | Execution time (seconds) |
|--------------|----------------|--------------------------|
| Binary       | Malan et al.   | 34                       |
|              | Yan and Shi    | 13.9                     |
|              | Eberle et al.  | 4.14                     |
|              | NanoECC        | 2.16                     |
|              | TinyECCK       | 1.14                     |
|              | Kargl et al.   | 0.83                     |
| Prime        | Wang and Li.   | 1.35                     |
|              | NanoECC        | 1.27                     |
|              | Gura et al.    | 0.87                     |
|              | Uhsadel et al. | 0.76                     |
|              | TinySA         | 0.745                    |

**Figure 32:** Timings for scalar multiplication (normalized for a clock frequency of 7.37MHz) [x]

The upper figure is basically indicating us the improvement in terms of efficiency of ECC in Wireless Sensor Networks.

The implementation developed by the authors of this study uses mixed additions from different modular operation algorithms (multiplications, reductions, inversions, etc.), in fact, the authors select fast algorithms for Elliptic Curve arithmetic in three situations: multiplying a random point  $P$  by a scalar  $k$  (1), multiplying the generator  $G$  by a scalar  $k$  (2) and simultaneously multiplying



two points  $P$  and  $Q$  by scalars  $k$  and  $l$  to obtain  $kP + lQ$  (3) (Basic mathematical details of the Elliptic Curves are found in chapter 2.6).

|                   | <b>Proposed</b> | <b>TinyECCK</b> | <b>Proposed</b> | <b>Kargl et al.</b> |
|-------------------|-----------------|-----------------|-----------------|---------------------|
| <b>Algorithm</b>  | C language      | C language      | Assembly        | Assembly            |
| Modular Squaring  | 1154 c          | 2729 c          | 570 c           | 663                 |
| Multiplication    | 9738 c          | 19670 c         | 4508 c          | 5057 c              |
| Modular reduction | 606 c           | 1904 c          | 430 c           | 433 c               |
| Inversion         | 243790 c        | 539132 c        | 81365 c         | –                   |
| $kP$ on Koblitz   | 0.67 s          | 1.14 s          | 0.32 s          | –                   |
| $kP$ on Generic   | 1.55 s          | –               | 0.74 s          | 0.83 s              |

**Figure 33:** Implementation results, comparison with two other implementations [x]

As presented in the upper figure, both the C language proposed implementation and the Assembled implementation perform better than the two most efficient ECC implementations shown in figure 32, TinyECCK and Kargl et al. Those results are the arithmetic mean of the timings measured on 50 consecutive executions of the algorithms. The executions consist of various operations, just as shown on the figure. Also, the results are shown whether in cycles (c) or seconds (s).

\*In this case (MICAz Mote sensor node equipped with an ATmega128 8-bit processor clocked at 7.3728MHz), arithmetic instructions with register operands cost 1 cycle and memory instructions or memory addressing cost 2 processing cycles.

But these improvements also carry other implications. For example, the implemented optimizations allow performance gains but advocate a collateral effect on memory consumption. As we will see in the following figure, memory requirements for code size and RAM memory for the different implementations are increased. This is something we will also see in the conclusions section, as we must find a threshold defining what shall be considered as a limitation in each design parameter.

|                                 | ROM memory | Static RAM | Stack RAM |
|---------------------------------|------------|------------|-----------|
| Proposed (Koblitz) – C          | 22092      | 1028       | 1207      |
| Proposed (Koblitz) – C+Assembly | 25802      | 1732       | 1207      |
| Proposed (Generic) – C          | 12848      | 881        | 682       |
| Proposed (Generic) – C+Assembly | 16218      | 1585       | 682       |
| TinyECCK (C-only)               | 5592       | –          | 618       |
| Kargl et a. (C+Assembly)        | 11264      | –          | –         |

**Figure 34:** ROM and RAM memory consumption of different implementations [x]

Last but not least, the author's implementation was tested with some executions of cryptographic protocols for key agreement and digital signatures. And the resulting performance was compared with two other protocols that are also being used in current Wireless Sensor Networks. Indeed, the two tested protocols are the following: **Key agreement**, which is employed in sensor networks for establishing symmetric keys that can be used for encryption or authentication and **Digital signatures**, employed for communication between the sensor nodes and the base stations where data must be made available to multiple applications and users. For

the first case, Key agreement, the Elliptic Curve Diffie & Hellman (ECDH) protocol was implemented, while for the second case, Digital signatures, the Elliptic Curve Digital Signature Algorithm (ECDSA) was used. In both cases, the public keys are supposed to be already loaded into the nodes, so the network initialization time is not taken into account.

| Curve     | C language |      |     | Assembly |      |     |
|-----------|------------|------|-----|----------|------|-----|
|           | Time       | ROM  | RAM | Time     | ROM  | RAM |
| NIST-K163 | 0.74       | 28.3 | 2.2 | 0.39     | 32.0 | 2.8 |
| NIST-B163 | 1.62       | 24.0 | 1.1 | 0.81     | 27.8 | 1.9 |
| NIST-K233 | 1.55       | 31.0 | 2.9 | 0.80     | 38.6 | 3.7 |
| NIST-B233 | 3.97       | 26.9 | 1.5 | 1.96     | 34.6 | 2.2 |

**Figure 35:** Performance for the Key agreement case [x]

| Curve     | C language   |      |     | Assembly     |      |     |
|-----------|--------------|------|-----|--------------|------|-----|
|           | Time (S + V) | ROM  | RAM | Time (S + V) | ROM  | RAM |
| NIST-K163 | 0.67 + 1.23  | 31.8 | 2.9 | 0.36 + 0.63  | 35.3 | 3.7 |
| NIST-B163 | 0.87 + 2.17  | 29.6 | 2.1 | 0.45 + 1.05  | 33.2 | 2.8 |
| NIST-K233 | 1.46 + 2.76  | 34.6 | 3.1 | 0.78 + 1.39  | 42.2 | 3.8 |
| NIST-B233 | 2.09 + 5.25  | 32.8 | 2.3 | 1.04 + 2.55  | 40.4 | 3.1 |

**Figure 36:** Performance for the Data signature case [x]

The *NIST-WXYZ* are different Elliptic Curves definitions published by NIST (with XYZ bits of security level). These curves are used to perform the test under various conditions and no only one Elliptic Curve.

\*Time (S+V): Signature + Verification time

\*ROM and RAM: Consumption values given in KB

These figures tell us how, for example, a digital signature can be computed and verified in 0.99 seconds at the 163-bit security level (NIST-K163, Data signature case, Assembly implementation) and in 2.17 seconds at the 233-bit security level (NIST-K233, Data signature case, Assembly implementation).

But still, *how do we know if these performance results are good enough for the IoT world? How can we compare it to the Ciphers performances?*

The work performed by David Malan on “*Crypto for Tiny Objects*” [vi] help us to discern it. This work was the first known implementation of elliptic curve cryptography for sensor networks. In fact, as we have just seen in the previous pages, the study “*Efficient Implementation of Elliptic Curve Cryptography in Wireless Sensors*” by Diego F. Aranha implements a most efficient solution than Malan’s work in the Binary field (figure 32). But, as just said, from Malan’s specific study, we are interested on the comparison of ECC performance with the performance parameters seen in previous sections.

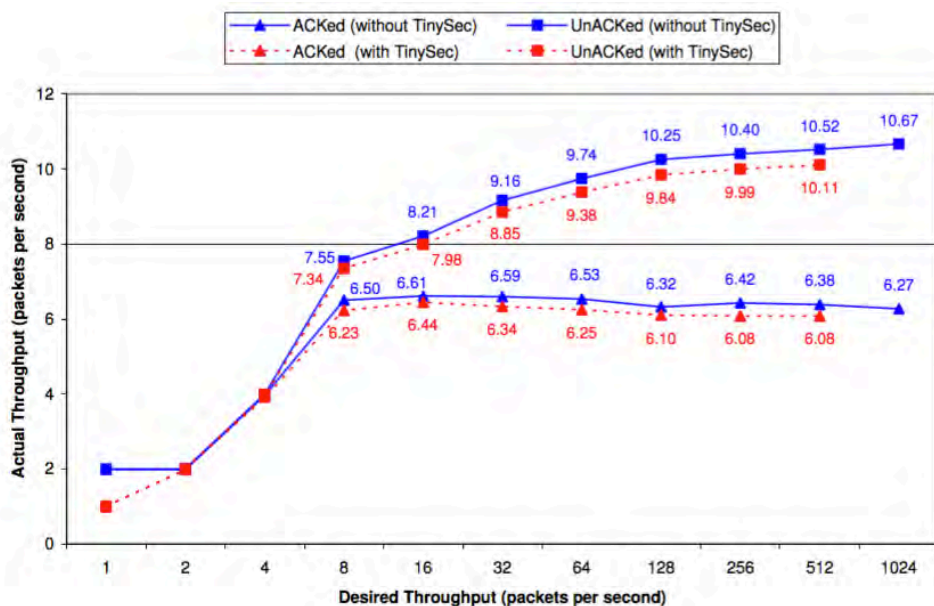
Malan’s work is performed with MICA2 mote, an 8-bit, 7.3828-MHz ATmega 128L processor, 4 kilobytes (KB) of SRAM, 128 KB of program space, 512 KB of EEPROM.

We will look at the results of an analysis of the MICA2's maximal throughput, (without and with *TinySec* (\*) enabled, something that is not important in our case as we are purely interested in the throughput levels). To add up, *TinyOS* (\*) is an embedded, component-based Operating Systems and platform for low-power wireless devices, such as those used in Wireless Sensor Networks (WSNs). It began as collaboration between the University of California, Berkeley, Intel Research, and Crossbow Technology, it was released as free and open-source software.

|                    | encrypt ()    | computeMAC () | Sum           |
|--------------------|---------------|---------------|---------------|
| Median             | 2,189 $\mu$ s | 3,038 $\mu$ s | 5,233 $\mu$ s |
| Mean               | 2,190 $\mu$ s | 3,049 $\mu$ s | 5,239 $\mu$ s |
| Standard Deviation | 3 $\mu$ s     | 281 $\mu$ s   | 281 $\mu$ s   |
| Standard Error     | 0 $\mu$ s     | 9 $\mu$ s     | 9 $\mu$ s     |

**Figure 37:** Latency ( $\mu$ s) of the encryption process [vi]

We can see how in the upper figure, the mean latency times of the encryption process are similar to those presented in **Figure 23**, proving that, for this implementation, latency equals in order to those presented algorithms.



**Figure 38:** Throughput information [vi]

Finally, as shown in figure 38, the throughput of sending packets of 29 bytes (1Kbits) on various scenarios achieves a level of 10.67 packets per second, which equals a total of 30.9Mbps. 10 times faster than the fastest design (which was actually not suitable for IoT cryptography) but, for example, 300 times faster than the MICKEY v2 algorithm that we saw in section 3.2.

\**TinySec* : Is a layer security architecture which was implemented in *TinyOS*

\**TinyOs*: is an embedded, component-based operating system and platform for low-power wireless devices.



## 4. IoT cryptographic algorithms simulation

As the final chapter of this study, after all the investigation about the various cryptographic algorithms had been carried out, it was intended to implement and test its functionality in a hardware device. Also the objective was to perform various transmissions of encrypted data in order to validate its performance.

The selected devices were the **Zolertia Z1 mote** and the **Sky mote**. The idea is to perform a comparison between two different Hardware pieces in order to see their performance differences. The Z1 is a general-purpose development platform for wireless sensor networks (WSN) designed for researchers and developers. It is equipped with a second generation MSP430F2617 low power microcontroller, which features a powerful 16-bit RISC CPU @16MHz clock speed, built-in clock factory calibration, 8KB RAM and a 92KB Flash memory. Also includes the well known CC2420 transceiver, IEEE 802.15.4 compliant, which operates at 2.4GHz with an effective data rate of 250Kbps. Z1 hardware selection guarantees the maximum efficiency and robustness with low energy cost. The Sky mote offers inter operability with many different IEEE 802.15.4 devices, as well as an ultra low current consumption, which fulfils the IoT study prerequisites. It has an effective data rate of 250Kbps CPU @2.4GHz clock speed, built-in clock factory calibration, 10KB RAM and a 48KB Flash memory

It was decided to work with a network simulator tool. The objective was to perform as much simulations as possible with different scenarios, so we would have a strong comparison basis in order to work. Different data message will be encrypted using the AES standard, the information will be sent to the other motes according to each scenario. While the process is running, various data values will be obtained, that way we will have enough information to carry on the mentioned comparison. Remembering that this comparison is performed under the thesis objectives, which are computational cost, data transmission rate and battery cost.

As said, the objective was to emulate the behaviour of the motes in a WSN environment, also, testing the code was important before working with the emulator mote devices.

Both the Zolertia Z1 and the Sky motes support various Operating Systems. The list is the following [xxxiii]:

- Contiki OS
- RIOT OS
- OpenWSN
- TinyOS
- MansOS

Among all of them, the Contiki OS is the one with more focus on Cryptography so it was the logical choice for us [xxxiv]. Contiki was developed by the Swedish Institute of computer science. It is defined as a lightweight, open source, highly portable and multitasking operating system used for embedded systems that are highly memory efficient. The memory usage for Contiki is about 2kb of RAM and 40Kb of ROM [xxxiv], which perfectly suits IoT devices specifications.

More precisely, the Contiki OS network simulator, named Cooja, will be used. This fact has the following advantage: It is ready not only to emulate the Z1 motes and Sky motes behaviour but also various other motes in case future comparisons would be intended to carry.

#### 4.1. Simulation premises - AES Standard

Before the start of the simulations we should set the following premise. **In order to work with different devices, we must select an encryption standard that is successfully emulated in various platforms/motes** (in this particular study it will be the Z1 and Sky motes). Between all the available standards the **AES** (Advanced Encryption System) was chosen.

The fact that we have worked with CLEFIA and MICKEY in the previous chapter doesn't imply that we can't select this standard (AES) for the simulation chapter. We must remember that those standards were chosen **in order to learn and study both the functioning and the features** of block ciphers and stream ciphers, so we could set a background work that helped us to know whether stream ciphers (with its own peculiarities) or block ciphers (also with its own features) fit the IoT world with more or less efficiency.

Also, it is important to note that, as mentioned in chapter 2, after the Advanced Encryption Standard was selected, many **block ciphers** with lightweight properties have been proposed. Among them, **CLEFIA** and **PRESENT** were well studied. We ended up choosing the CLEFIA block cipher to conduct the study, but both of them are generated from the AES standard. This is also important, as the following simulations could serve as basis for future work (using Contiki emulator) thanks to the fact of using AES.

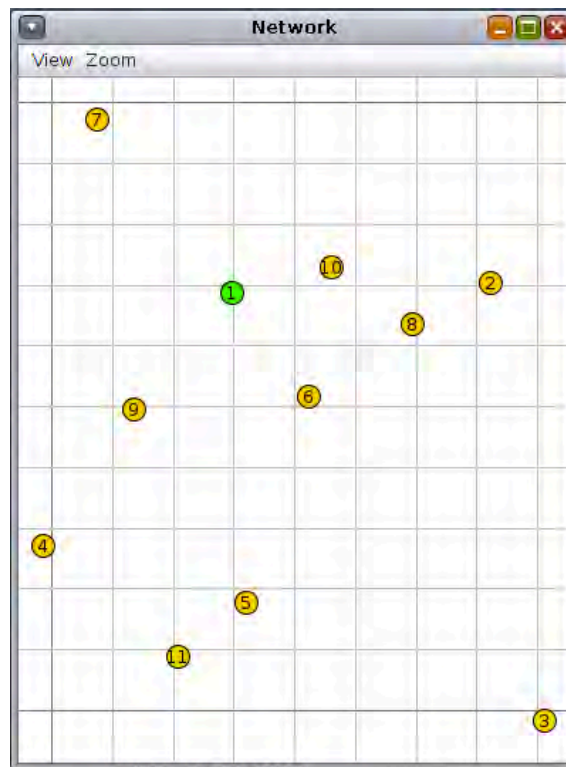
We should also remember that after all the gathered and analysed information, this **simulation chapter will help us discern, under the same scenario and algorithm usage, which case suits best the IoT world** (in terms of computational cost, data transmission rate and power consumption) in a well-defined closed framework. That is why, in the various simulation cases, the mote placement will be the same independently of the mote type and the usage or not of encryption.

Finally, remembering that for AES standard, both high speed and low RAM requirements were criteria of the AS selection process. As the chosen algorithm, AES performed well on a wide variety of hardware, from 8-bit smart cards to high-performance computers [xix]. More standard information (related to simulation purposes) will be displayed in the following sub-chapters.



## 4.2. Simulation 1 – Sky mote – Sink-Sender - WSN without cryptographic algorithms

The first step was to design and implement a base environment. This environment will be used as the starting point to the rest of the simulations.



**Figure 39: First IoT WSN**

This first scenario is designed with 1 node (mote) acting as a server and 10 nodes (motes) doing the functionality of senders. The server module used in this simulation is an UDP sink. The Contiki's function representing this mote is called `udp-sink.c`. On the other hand, the sender motes used in this simulation are UDP clients, represented by the `udp-sender.c` function. Those client motes send UDP packets to the server, so connection can be established.

```
00:01.171 ID:3 Rime started with address 0.18.116.3.0.3.3.3
00:01.180 ID:3 MAC 00:12:74:03:00:03:03:03 Contiki-2.6-2450-geaa8760 started. Node id is set to 3.
00:01.189 ID:3 nullsec CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26, CCA threshold -45
00:01.200 ID:3 Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7403:0003:0303
00:01.203 ID:3 Starting 'UDP client process' 'collect common process'
00:01.206 ID:3 UDP client process started
00:01.211 ID:3 Client IPv6 addresses: aaaa::212:7403:3:303
00:01.214 ID:3 fe80::212:7403:3:303
00:01.219 ID:3 Created a connection with the server :: local/remote port 8775/5688
```

**Figure 40: Client initialization**

As it can be seen in the upper figure, each client process is initialized using this structure. An ID and MAC address is assigned to the sender mote, after that the channel parameters are also generated and, finally, an IP address (this is an example using IPv6 address) is assigned to this mote. Once this device has correctly started it creates a connection with the server mote using the ports shown in the figure.

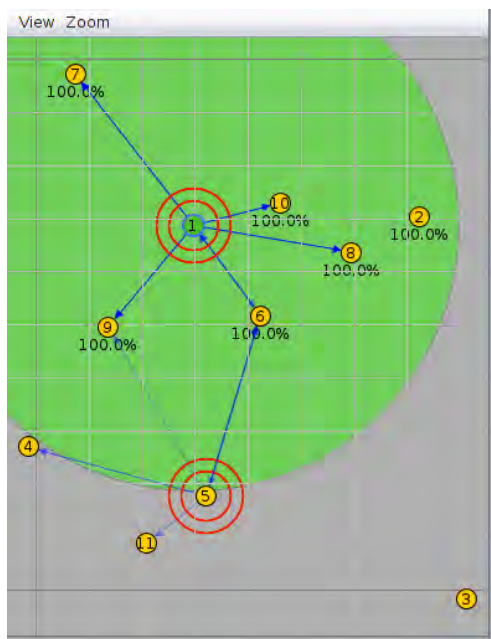


Figure 41: WSN packets transmission

While the packet transmission is going on, the sensor data collector tool is used to store the results. In this case, the following sensor map is generated:

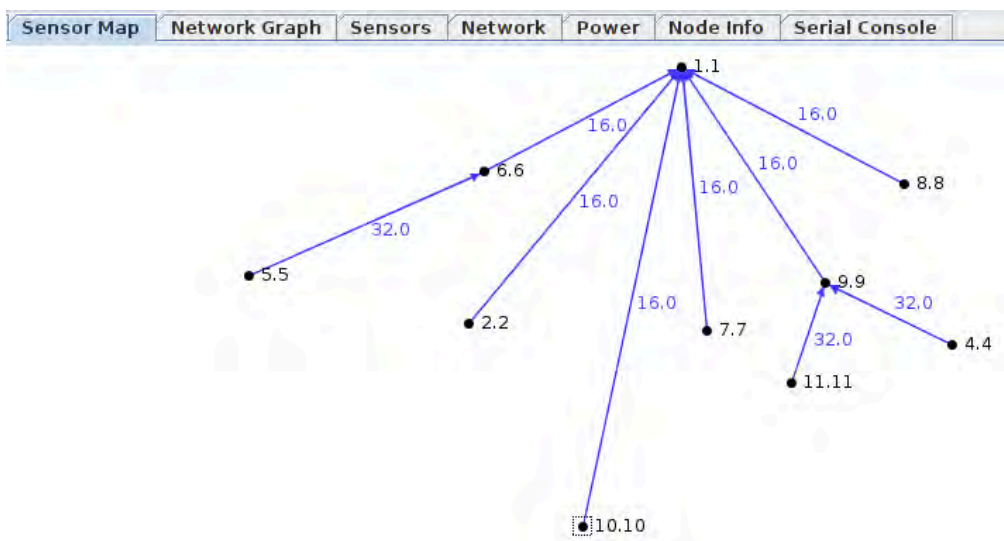


Figure 42: Sensor map of the first simulation

It is interesting to see how the mote with ID number 3 is not represented in this map. This is because of the power limitation of the motes. As we will see in the following figure, this mote is too far away from the other senders, so the packets get lost and never reach the other motes. As it cannot reach the server, this mote is never initialized thus the server doesn't see it and does not integrate the wireless network. It can also be seen how this representation shows that motes number 4, 5 and 11 need to perform one hop through motes 6 and 9 in order to transmit its information to the server. This fact is perfectly indicated by the ETX value of 32 in all three cases. The Expected Transmission Count is a measure of the quality of a path between two nodes in a wireless packet data network.



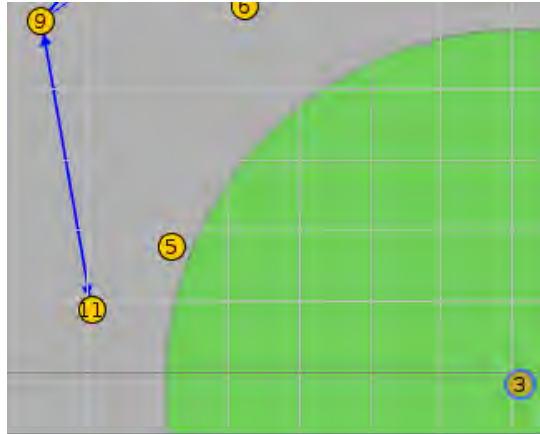


Figure 43: Mote 3 network isolation

The simulator also gathers multiple sensor information, such as:

| Node Control |              | Sensor Map   |              |              | Network Graph  |               |              | Sensors              |              | Network      |              | Power        |                | Node Info    |                      | Serial Console    |  |
|--------------|--------------|--------------|--------------|--------------|----------------|---------------|--------------|----------------------|--------------|--------------|--------------|--------------|----------------|--------------|----------------------|-------------------|--|
| Node         | Received     | Dups         | Lost         | Hops         | Rtmetric       | ETX           | Churn        | Beacon Interval      | Reboots      | CPU Power    | LPM Power    | Listen Power | Transmit Power | Power        | On-time              | Listen Duty Cycle |  |
| 1.1          | 0            | 0            | 0            | 0.000        | 0.000          | 0.000         | 0            |                      | 0            | 0.000        | 0.000        | 0.000        | 0.000          | 0.000        |                      | 0.000             |  |
| 2.2          | 5            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 4 min, 48 sec        | 0            | 0.382        | 0.152        | 0.446        | 0.099          | 1.079        | 0 min, 57 sec        | 0.743             |  |
| 4.4          | 5            | 0            | 0            | 2.000        | 768.000        | 32.000        | 0            | 4 min, 22 sec        | 0            | 0.370        | 0.152        | 0.486        | 0.191          | 1.199        | 1 min, 06 sec        | 0.810             |  |
| 5.5          | 5            | 0            | 0            | 2.000        | 768.000        | 32.000        | 0            | 4 min, 48 sec        | 0            | 0.390        | 0.152        | 0.492        | 0.208          | 1.241        | 1 min, 04 sec        | 0.820             |  |
| 6.6          | 5            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 4 min, 48 sec        | 0            | 0.420        | 0.151        | 0.481        | 0.096          | 1.148        | 1 min, 03 sec        | 0.802             |  |
| 7.7          | 6            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 6 min, 11 sec        | 0            | 0.342        | 0.153        | 0.431        | 0.085          | 1.011        | 1 min, 11 sec        | 0.718             |  |
| 8.8          | 6            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 5 min, 27 sec        | 0            | 0.372        | 0.152        | 0.446        | 0.091          | 1.061        | 1 min, 16 sec        | 0.744             |  |
| 9.9          | 6            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 6 min, 11 sec        | 0            | 0.418        | 0.151        | 0.495        | 0.083          | 1.147        | 1 min, 15 sec        | 0.825             |  |
| 10.10        | 5            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 5 min, 01 sec        | 0            | 0.424        | 0.151        | 0.484        | 0.142          | 1.201        | 1 min, 13 sec        | 0.807             |  |
| 11.11        | 5            | 0            | 0            | 2.000        | 776.200        | 32.000        | 0            | 5 min, 40 sec        | 0            | 0.369        | 0.152        | 0.476        | 0.156          | 1.153        | 0 min, 57 sec        | 0.793             |  |
| <b>Avg</b>   | <b>5.333</b> | <b>0.000</b> | <b>0.000</b> | <b>1.333</b> | <b>598.244</b> | <b>21.333</b> | <b>0.000</b> | <b>5 min, 15 sec</b> | <b>0.000</b> | <b>0.387</b> | <b>0.152</b> | <b>0.471</b> | <b>0.128</b>   | <b>1.138</b> | <b>1 min, 07 sec</b> | <b>0.785</b>      |  |

Figure 44: Node Info table (I)

| Transmit Duty Cycle | Avg Inter-packet Time | Min Inter-packet Time | Max Inter-packet Time |
|---------------------|-----------------------|-----------------------|-----------------------|
| 0.000               |                       |                       |                       |
| 0.187               | 0 min, 45 sec         | 0 min, 05 sec         | 1 min, 24 sec         |
| 0.359               | 0 min, 47 sec         | 0 min, 25 sec         | 1 min, 35 sec         |
| 0.391               | 0 min, 43 sec         | 0 min, 31 sec         | 1 min, 27 sec         |
| 0.180               | 0 min, 42 sec         | 0 min, 30 sec         | 1 min, 18 sec         |
| 0.160               | 0 min, 47 sec         | 0 min, 07 sec         | 1 min, 28 sec         |
| 0.171               | 0 min, 50 sec         | 0 min, 28 sec         | 1 min, 31 sec         |
| 0.156               | 0 min, 51 sec         | 0 min, 51 sec         | 1 min, 15 sec         |
| 0.268               | 0 min, 55 sec         | 0 min, 57 sec         | 1 min, 28 sec         |
| 0.294               | 0 min, 53 sec         | 0 min, 54 sec         | 1 min, 18 sec         |
| <b>0.241</b>        | <b>0 min, 48 sec</b>  | <b>0 min, 32 sec</b>  | <b>1 min, 24 sec</b>  |

Figure 45: Node Info table (II)

We will recover this table when comparing the diverse simulations to be performed using Cooja. Among others, it can be seen how the average power consumed by the motes is 1'138mW (a sum of CPU power, LPM Power, Listen Power and Transmit Power), the average Inter-packet Time is 48 seconds and the average Transmit Duty Cycle, the fraction of one period in which a the system is active, is 0'241. It also shows information about the average Beacon interval, one of the management frames in IEEE 802.11 based WLANs. It contains all the information about the network. Beacon frames are transmitted periodically, they serve to announce the presence of a wireless LAN and to synchronise the members of the service set.

Power distribution on each mote as well as Radio Duty Cycle figures are shown next:

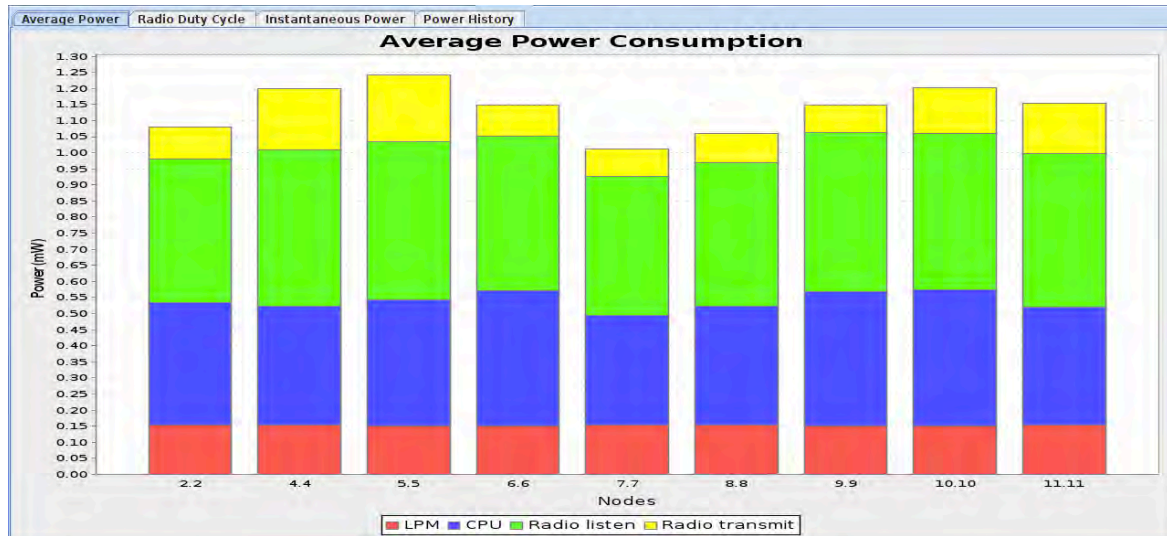


Figure 46: Power distribution per mote

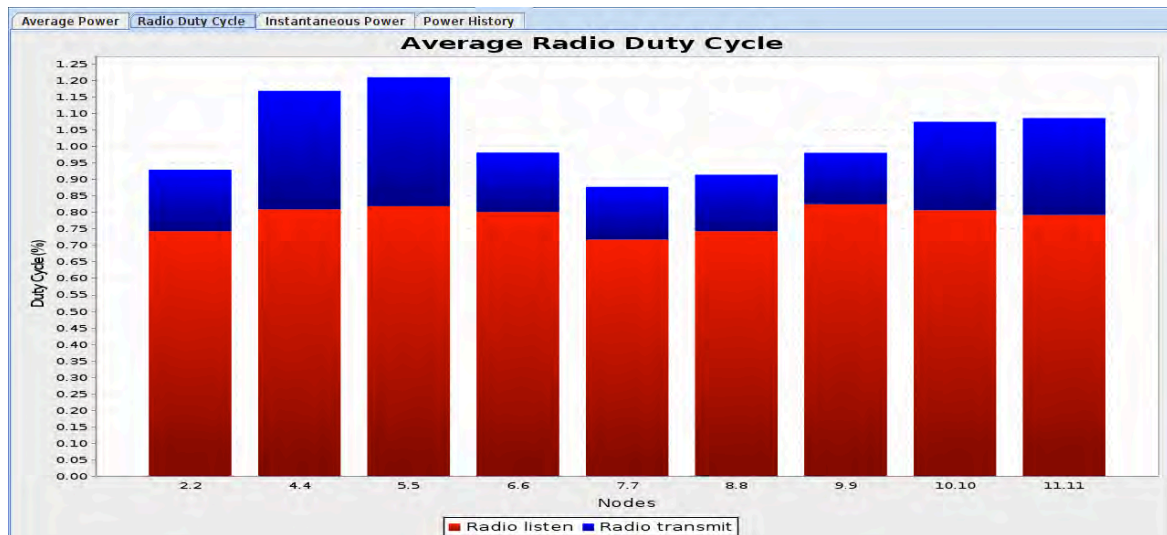


Figure 47: Radio Duty Cycle per mote

### 4.3. Simulation 2 – Z1 mote – Sink-Sender - WSN without cryptographic algorithms

In this scenario we will repeat the first simulation, performed with the Sky mote, but now using the Z1 mote. This fact helps us to expand the sample analysis, as we have the same scenario simulated two times using different hardware equipment.



Figure 48: Z1 IoT WSN

This second scenario is designed using the same exact format as scenario presented in section 4.2. One node (mote) will act as a sink server and 10 nodes (motes) will perform the sender functionality.

```

00:02.977 ID:9 Rime started with address 193.12.0.0.0.0.9
00:02.993 ID:9 MAC c1:0c:00:00:00:00:09 Contiki-2.6-2450-geaa8760 started. Node id is set to 9.
00:03.003 ID:9 CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
00:03.018 ID:9 Tentative link-local IPv6 address fe80:0000:0000:0000:c30c:0000:0000:0009
00:03.026 ID:9 Starting 'UDP client process' 'collect common process'
00:03.031 ID:9 UDP client process started
00:03.037 ID:9 Client IPv6 addresses: aaaa::c30c:0:0:9
00:03.041 ID:9 fe80::c30c:0:0:9
00:03.051 ID:9 Created a connection with the server :: local/remote port 8775/5688

```

Figure 49: Z1 client initialization

As it can be seen by comparing figure 39 and 48, in both cases one mote has been placed in an “isolated” area in order to prove the correct initialization of the environment. If everything goes right, mote number 3 shall never appear in the sensor map.

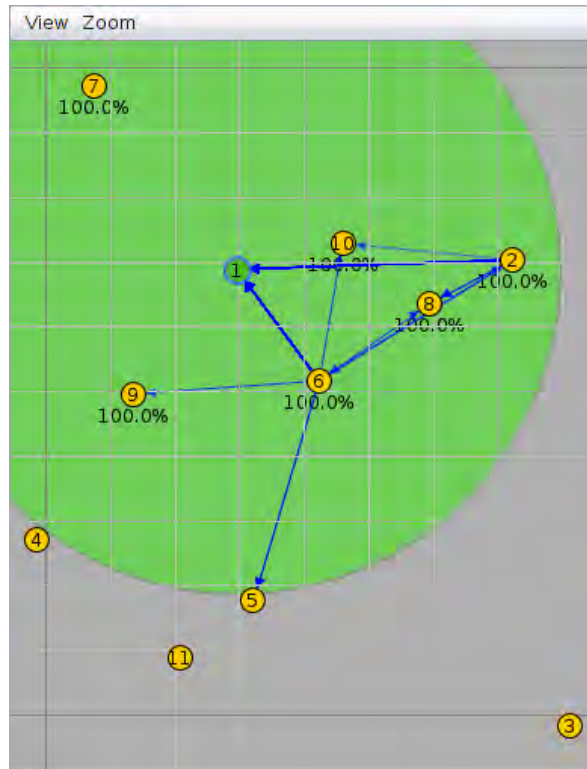


Figure 50: Z1 motes packet transmission

While the packet transmission is going on, the sensor map is collected like the previous case:

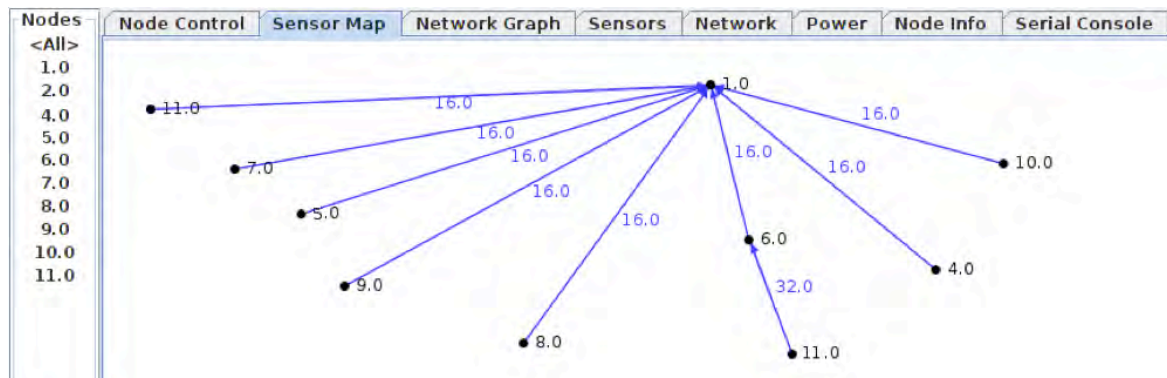


Figure 51: Z1 Sensor map

So, as expected, mote 3 does not appear in the sensor map, as its transmission power is not big enough to communicate with the sink mote (number 1). Also, it is not close enough to get its synchronization performed hoping to another mote like happens with mote number 2.

As just commented, it can also be seen how this representation shows that mote number 11 needs to perform one hop through mote 6 in order to transmit its information to the server. This fact is perfectly indicated by the ETX value of 32 for this case.

Just like in the previous case, multiple sensor information is gathered during the simulation time, such as the following:

| Node Control |              | Sensor Map   |              |              | Network Graph  |               |              | Sensors              |              | Network      |              | Power        |                | Node Info    |                  | Serial Console    |                     |
|--------------|--------------|--------------|--------------|--------------|----------------|---------------|--------------|----------------------|--------------|--------------|--------------|--------------|----------------|--------------|------------------|-------------------|---------------------|
| Node         | Received     | Dups         | Lost         | Hops         | Rtmetric       | ETX           | Churn        | Beacon Interval      | Reboots      | CPU Power    | LPM Power    | Listen Power | Transmit Power | Power        | On-time          | Listen Duty Cycle | Transmit Duty Cycle |
| 1.0          | 0            | 0            | 0            | 0.000        | 768.000        | 0.000         | 0            | 5 min, 16 sec        | 0            | 0.000        | 0.000        | 0.000        | 0.000          | 0.000        | 0.000            | 0.000             | 0.000               |
| 2.0          | 6            | 0            | 0            | 2.000        | 512.000        | 16....        | 0            | 5 min, 05 sec        | 0            | 0.069        | 0.161        | 0.251        | 0.180          | 0.662        | 1 min,...        | 0.418             | 0.340               |
| 3.0          | 6            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 5 min, 05 sec        | 0            | 0.094        | 0.161        | 0.223        | 0.100          | 0.577        | 1 min,...        | 0.371             | 0.188               |
| 4.0          | 5            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 4 min, 48 sec        | 0            | 0.116        | 0.160        | 0.252        | 0.097          | 0.624        | 1 min,...        | 0.420             | 0.182               |
| 6.0          | 6            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 6 min, 11 sec        | 0            | 0.093        | 0.161        | 0.233        | 0.092          | 0.579        | 1 min,...        | 0.388             | 0.174               |
| 7.0          | 5            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 5 min, 14 sec        | 0            | 0.087        | 0.161        | 0.227        | 0.106          | 0.580        | 0 min,...        | 0.378             | 0.199               |
| 8.0          | 5            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 5 min, 40 sec        | 0            | 0.116        | 0.160        | 0.237        | 0.114          | 0.628        | 0 min,...        | 0.396             | 0.215               |
| 9.0          | 6            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 5 min, 49 sec        | 0            | 0.124        | 0.160        | 0.241        | 0.138          | 0.663        | 1 min,...        | 0.402             | 0.259               |
| 10.0         | 6            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 5 min, 27 sec        | 0            | 0.095        | 0.161        | 0.229        | 0.101          | 0.586        | 1 min,...        | 0.382             | 0.190               |
| 11.0         | 5            | 0            | 0            | 1.000        | 512.000        | 16....        | 0            | 4 min, 22 sec        | 0            | 0.083        | 0.161        | 0.215        | 0.092          | 0.551        | 1 min,...        | 0.359             | 0.173               |
| <b>Avg</b>   | <b>5.556</b> | <b>0.000</b> | <b>0.000</b> | <b>1.111</b> | <b>540.444</b> | <b>17....</b> | <b>0.000</b> | <b>5 min, 19 sec</b> | <b>0.000</b> | <b>0.098</b> | <b>0.161</b> | <b>0.234</b> | <b>0.113</b>   | <b>0.606</b> | <b>1 min,...</b> | <b>0.390</b>      | <b>0.213</b>        |

Figure 52: Node Info table (I)

| Transmit Duty Cycle | Avg Inter-packet Time | Min Inter-packet Time | Max Inter-packet Time |
|---------------------|-----------------------|-----------------------|-----------------------|
| 0.000               |                       |                       |                       |
| 0.340               | 0 min, 50 sec         | 0 min, 19 sec         | 1 min, 42 sec         |
| 0.188               | 0 min, 49 sec         | 0 min, 39 sec         | 1 min, 26 sec         |
| 0.182               | 0 min, 45 sec         | 0 min, 19 sec         | 1 min, 28 sec         |
| 0.174               | 0 min, 50 sec         | 0 min, 43 sec         | 1 min, 23 sec         |
| 0.199               | 0 min, 46 sec         | 0 min, 08 sec         | 1 min, 41 sec         |
| 0.215               | 0 min, 52 sec         | 0 min, 48 sec         | 1 min, 17 sec         |
| 0.259               | 0 min, 42 sec         | 0 min, 12 sec         | 1 min, 17 sec         |
| 0.190               | 0 min, 45 sec         | 0 min, 27 sec         | 1 min, 37 sec         |
| 0.173               | 0 min, 46 sec         | 0 min, 47 sec         | 1 min, 03 sec         |
| <b>0.213</b>        | <b>0 min, 47 sec</b>  | <b>0 min, 29 sec</b>  | <b>1 min, 26 sec</b>  |

Figure 53: Node Info table (II)

As commented in the previous chapter, this table will be recovered when comparing the diverse simulations performed using Cooja. Among others, it can be seen how the average power consumed by the Z1 motes is 0'606mW (a sum of CPU power, LPM Power, Listen Power and Transmit Power), which is significantly lower than the power consumed by the Sky motes. In this case the average Inter-packet time is 47 seconds, 1 second lower than the previous simulation, and the average Transmit Duty Cycle, the fraction of one period in which a the system is active, is 0'213 which is also lower than the Sky motes case. It also shows information about the average Beacon interval

Power distribution on each mote as well as Radio Duty Cycle figures are shown in the next page.



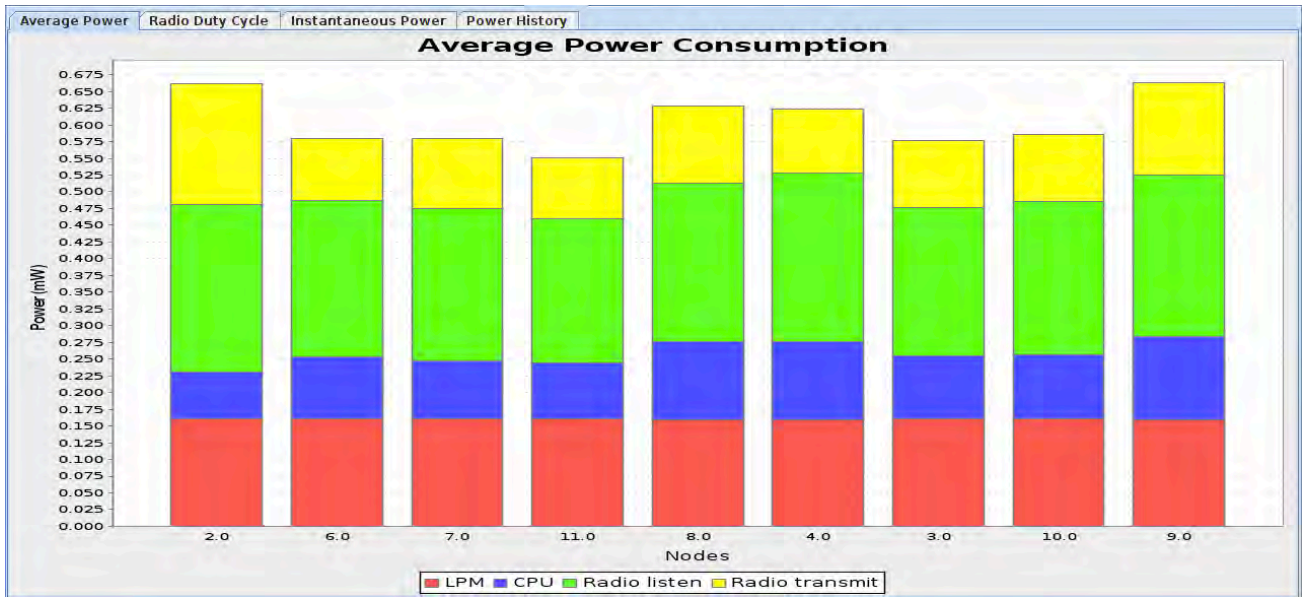


Figure 54: Power consumption per mote – Z1

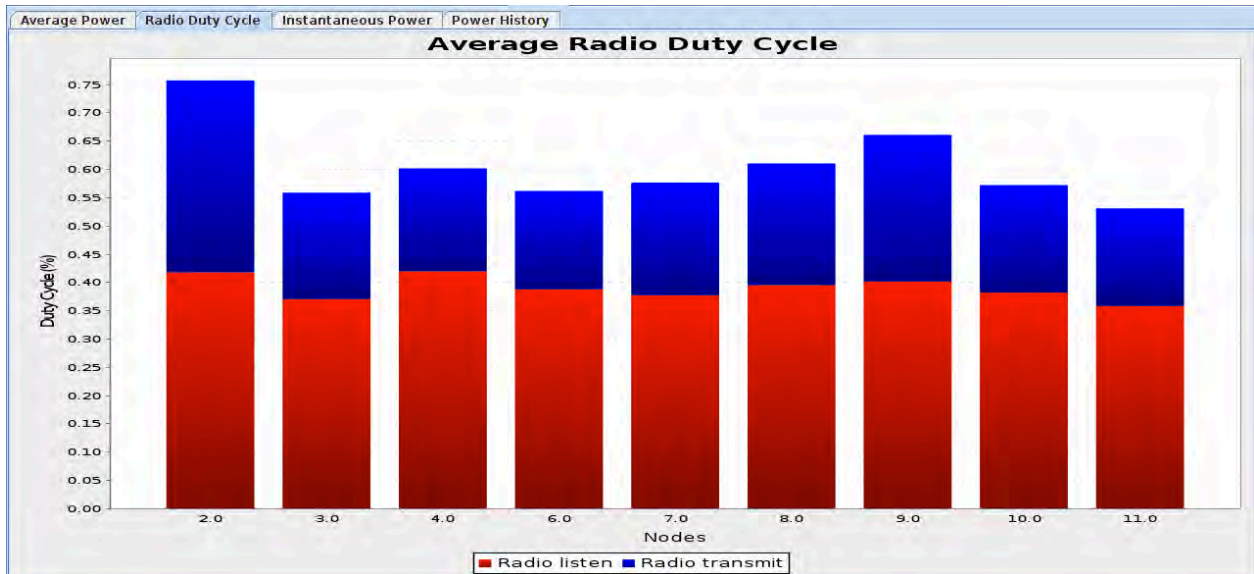


Figure 55: Radio duty cycle per mote – Z1

#### 4.4. Enabling encryption

Contiki has LLSEC (link-layer security) layer, which is a hardware independent layer. It uses a generic AES driver API instead of directly accessing the hardware. There are multiple AES drivers implemented in Contiki - a software-only version and a couple of hardware accelerated ones, including for CC2420 (the radio chip on Tmote Sky, designed for low power and low voltage wireless applications).

The “problem” with Cooja is that the HW acceleration feature of CC2420 is not implemented in the MSPsim emulator that Cooja uses (The MSPsim is a Java-based instruction level emulator of the MSP430 series microprocessor and emulation of some sensor networking platforms). That is why Hardware acceleration is not going to work in Cooja as opposed to real Tmote Sky nodes (for example); the software-based AES driver in configuration must be explicitly selected [xxxvi] using this code line:

```
#define AES_128_CONF aes_128_driver
```

As indicated in the github repository [xxxvii]:

`noncoresec` is a noncompromise-resilient 802.15.4 security implementation, which uses a network-wide key. Adding these lines to the `project_conf.h` file enables the `noncoresec` security implementation:

```
#undef LLSEC802154_CONF_ENABLED
#define LLSEC802154_CONF_ENABLED          1
#undef NETSTACK_CONF_FRAMER
#define NETSTACK_CONF_FRAMER             noncoresec_framer
#undef NETSTACK_CONF_LLSEC
#define NETSTACK_CONF_LLSEC               noncoresec_driver
#undef NONCORESEC_CONF_SEC_LVL
#define NONCORESEC_CONF_SEC_LVL           1
```

`NONCORESEC_CONF_SEC_LVL` defines the length of MICs and whether encryption is enabled or not. This parameter corresponds to the IEEE 802.15.4 framer security levels, with numerical values from 0x0 to 0x07.

The possible parameter values are the following:

- 0x00 No security Data is not encrypted. Data authenticity is not validated.
- 0x01 AES-CBC-MAC-32 MIC-32 Data is not encrypted. Data authenticity is validated.
- 0x02 AES-CBC-MAC-64 MIC-64 Data is not encrypted. Data authenticity is validated.
- 0x03 AES-CBC-MAC-128 MIC-128 Data is not encrypted. Data authenticity is validated.
- 0x04 AES-CTR ENC Data is encrypted. Data authenticity is not validated.
- 0x05 AES-CCM-32 AES-CCM-32 Data is encrypted. Data authenticity is validated.
- 0x06 AES-CCM-64 AES-CCM-64 Data is encrypted. Data authenticity is validated.
- 0x07 AES-CCM-128 AES-CCM-128 Data is encrypted. Data authenticity is validated.

So in our case we will be interested in the following values:

```
#define NONCORESEC_CONF_SEC_LVL 0x4  
#define NONCORESEC_CONF_SEC_LVL 0x5  
#define NONCORESEC_CONF_SEC_LVL 0x6  
#define NONCORESEC_CONF_SEC_LVL 0x7
```

It is also important to notice that there is no support for hardware-accelerated asymmetric encryption on sensor nodes. Also, there are no software-based implementations for that in mainline Contiki; there is no support (yet) for end-to-end security in general in this OS, as opposed to link-layer security.

This shows us how AES can be emulated with and without data encryption with various modes of operation for cryptographic block ciphers. It is an authenticated encryption algorithm designed to provide both authentication and confidentiality. CCM mode is only defined for block ciphers with a block length of 128 bits that will be then, the key size used in those simulations.

But before enabling encryption between the nodes, a broadcast messaging scenario will be first simulated, as it is more similar to a random IoT application in which the different nodes may be sending information to each other during the application work time.



### 4.5. Simulations 3/4 – Sky mote, Z1 mote – Broadcast WSN simulation no encryption

In this simulation we set an UDP-RPL broadcast example using the same network structure:

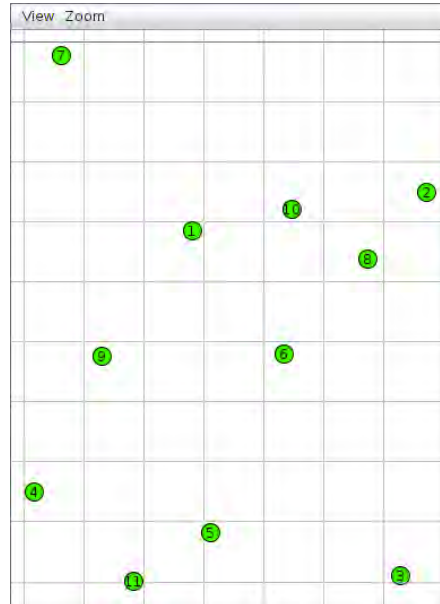


Figure 56: Broadcast WSN

In this scenario we use the function `broadcast-example.c`, which emulates the Sky mote behaviour, broadcasting packets to the other motes.

The screenshot displays a simulation control window with buttons for 'Start', 'Pause', 'Step', and 'Reload'. The 'Time' is 01:31.048 and 'Speed' is ---. Below this is a 'Notes' window and a 'Note output' window. The main window shows a network diagram with nodes 1-11 and their connection status (e.g., 100.0%). A log window shows the following data:

| Time      | Mote  | Message   |
|-----------|-------|---|
| 01:26.967 | ID:11 | 5479 P 193.12 20 26115 1350089 4718 5108 0 0 818 64715 0 192 0 0 (radio 0.71% / 0.29% tx 0... |
| 01:27.013 | ID:2  | 5479 P 193.12 20 26011 1350197 4721 4820 0 0 821 64712 0 192 0 0 (radio 0.69% / 0.29% tx 0... |
| 01:27.027 | ID:6  | 5479 P 193.12 20 26315 1349887 4721 5475 0 0 821 64712 0 192 0 0 (radio 0.74% / 0.29% tx 0... |
| 01:27.125 | ID:4  | 5479 P 193.12 20 25697 1350514 3698 4807 0 0 821 64712 0 192 0 0 (radio 0.61% / 0.29% tx 0... |
| 01:27.159 | ID:1  | 5479 P 193.12 20 25939 1350270 4716 5293 0 0 822 64711 0 192 0 0 (radio 0.72% / 0.29% tx 0... |
| 01:27.178 | ID:7  | 5479 P 193.12 20 25508 1350700 4724 4661 0 0 821 64712 0 192 0 0 (radio 0.68% / 0.29% tx 0... |
| 01:27.333 | ID:10 | 5479 P 193.12 20 26157 1350039 4719 5633 0 0 821 64712 0 192 0 0 (radio 0.75% / 0.29% tx 0... |
| 01:27.492 | ID:5  | 5479 P 193.12 20 26086 1350124 4720 5357 0 0 821 64712 0 192 0 0 (radio 0.73% / 0.29% tx 0... |
| 01:27.676 | ID:3  | 5479 P 193.12 20 25370 1350841 4723 4822 0 0 818 64715 0 192 0 0 (radio 0.69% / 0.29% tx 0... |
| 01:30.849 | ID:8  | 5735 P 193.12 21 26939 1414802 4716 5207 0 0 804 64728 0 192 0 0 (radio 0.68% / 0.29% tx 0... |
| 01:30.852 | ID:8  | Sending broadcast   |
| 01:30.947 | ID:9  | Data received on port 1234 from port 1234 with length 4                                       |
| 01:30.967 | ID:11 | 5735 P 193.12 21 26920 1414817 4718 5300 0 0 804 64728 0 192 0 0 (radio 0.69% / 0.29% tx 0... |
| 01:30.970 | ID:11 | Sending broadcast   |
| 01:30.971 | ID:2  | Data received on port 1234 from port 1234 with length 4                                       |
| 01:30.990 | ID:6  | Data received on port 1234 from port 1234 with length 4                                       |
| 01:31.013 | ID:2  | 5735 P 193.12 21 27004 1414737 4721 5066 0 0 992 64540 0 246 0 0 (radio 0.67% / 0.37% tx 0... |
| 01:31.027 | ID:6  | 5735 P 193.12 21 27313 1414423 4721 5788 0 0 997 64536 0 313 0 0 (radio 0.72% / 0.47% tx 0... |

At the bottom, a timeline shows the sequence of events for nodes 1 through 6.

Figure 57: Broadcast WSN simulation packet sending

| Time      | Mote  | Message   |
|-----------|-------|---|
| 00:21.392 | ID:10 | Sending broadcast                                       |
| 00:21.420 | ID:6  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.445 | ID:3  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.449 | ID:7  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.492 | ID:8  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.503 | ID:9  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.510 | ID:5  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.517 | ID:4  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.783 | ID:8  | Sending broadcast                                       |
| 00:21.820 | ID:3  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.855 | ID:10 | Data received on port 1234 from port 1234 with length 4 |
| 00:21.881 | ID:9  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.888 | ID:5  | Data received on port 1234 from port 1234 with length 4 |
| 00:21.895 | ID:4  | Data received on port 1234 from port 1234 with length 4 |

Figure 58: Mote output broadcast data

As it can be seen in the upper figure the Sky motes are sending and receiving data alternatively as part of the broadcast process. Let's see the power consumption of these activities in order to compare it when encryption is enabled. In order to do so, we have to modify the function `broadcast-example.c` by adding the `powertrace` functionality.

To include the functionality of PowerTrace in the code, a new function called `broadcast-example-2.c` is created by simply add the following line after `PROCESS_BEGIN()`; for the code.

```
powertrace_start(CLOCK_SECOND * 2);
```

The header file must be included:

```
#include "powertrace.h"
```

After that, in the file called "Makefile" in the working folder, we have to add the following line:

```
APPS+=powertrace
```

After running the simulation, the following data on the mote output window is generated:

| Time      | Mote  | Message   |
|-----------|-------|---|
| 01:02.572 | ID:2  | 7942 P 0.18 30 62786 1968753 10624 14802 0 13312 6045 59449 2612 487 0 405 (radio 1.25% / 4.73% tx 0.52% / 3.98% listen 0.72% / 0.74%)  |
| 01:02.587 | ID:6  | 7942 P 0.18 30 64993 1966552 10621 17390 0 14176 6263 59231 2612 1082 0 555 (radio 1.37% / 5.64% tx 0.52% / 3.98% listen 0.85% / 1.65%) |
| 01:02.683 | ID:4  | 7942 P 0.18 30 61010 1970532 8003 18037 0 14169 2092 63400 0 1377 0 717 (radio 1.28% / 2.10% tx 0.39% / 0.00% listen 0.88% / 2.10%)     |
| 01:02.718 | ID:1  | 7942 P 0.18 30 62611 1968936 10625 14760 0 13314 6011 59483 2612 487 0 405 (radio 1.24% / 4.73% tx 0.52% / 3.98% listen 0.72% / 0.74%)  |
| 01:02.736 | ID:7  | 7942 P 0.18 30 59330 1972202 8006 16499 0 13707 1588 63903 0 774 0 585 (radio 1.20% / 1.18% tx 0.39% / 0.00% listen 0.81% / 1.18%)      |
| 01:02.891 | ID:10 | 7942 P 0.18 30 60798 1970769 8009 18120 0 14669 1650 63842 0 954 0 554 (radio 1.28% / 1.45% tx 0.39% / 0.00% listen 0.89% / 1.45%)      |
| 01:03.044 | ID:9  | 7942 P 0.18 30 65706 1965827 10621 18042 0 15176 6109 59384 2612 883 0 771 (radio 1.41% / 5.33% tx 0.52% / 3.98% listen 0.88% / 1.34%)  |
| 01:03.051 | ID:5  | 7942 P 0.18 30 64196 1967354 10621 17202 0 15041 6207 59283 2612 898 0 785 (radio 1.36% / 5.35% tx 0.52% / 3.98% listen 0.84% / 1.37%)  |
| 01:03.234 | ID:3  | 7942 P 0.18 30 60768 1970788 8006 18005 0 14776 1602 63889 0 802 0 772 (radio 1.28% / 1.22% tx 0.39% / 0.00% listen 0.88% / 1.22%)      |
| 01:03.271 | ID:2  | Sending broadcast   |
| 01:03.322 | ID:3  | Data received on port 1234 from port 1234 with length 4   |
| 01:03.326 | ID:7  | Data received on port 1234 from port 1234 with length 4   |
| 01:04.408 | ID:8  | 8198 P 0.18 31 65824 2031226 10712 18121 0 14250 1419 64071 0 432 0 432 (radio 1.37% / 0.65% tx 0.51% / 0.00% listen 0.86% / 0.65%)     |
| 01:04.572 | ID:2  | 8198 P 0.18 31 68923 2028109 13318 15285 0 13716 6134 59356 2694 483 0 404 (radio 1.36% / 4.85% tx 0.63% / 4.11% listen 0.72% / 0.73%)  |
| 01:04.585 | ID:6  | 8198 P 0.18 31 66475 2030562 10621 17822 0 14608 1479 64010 0 432 0 432 (radio 1.35% / 0.65% tx 0.50% / 0.00% listen 0.84% / 0.65%)     |
| 01:04.683 | ID:4  | 8198 P 0.18 31 62444 2034590 8003 18469 0 14601 1431 64058 0 432 0 432 (radio 1.26% / 0.65% tx 0.38% / 0.00% listen 0.88% / 0.65%)      |
| 01:04.717 | ID:1  | 8198 P 0.18 31 64079 2032960 10625 15192 0 13746 1465 64024 0 432 0 432 (radio 1.23% / 0.65% tx 0.50% / 0.00% listen 0.72% / 0.65%)     |
| 01:04.736 | ID:7  | 8198 P 0.18 31 61010 2036013 8006 17127 0 14315 1677 63811 0 628 0 608 (radio 1.19% / 0.95% tx 0.38% / 0.00% listen 0.81% / 0.95%)      |
| 01:05.017 | ID:10 | 8214 P 0.18 31 66807 2034351 10627 18637 0 15101 6006 63582 2618 517 0 432 (radio 1.39% / 4.50% tx 0.50% / 3.76% listen 0.88% / 0.74%)  |
| 01:05.043 | ID:9  | 8198 P 0.18 31 67269 2029755 10621 18656 0 15581 1560 63928 0 614 0 405 (radio 1.39% / 0.93% tx 0.50% / 0.00% listen 0.88% / 0.93%)     |
| 01:05.050 | ID:5  | 8198 P 0.18 31 65766 2031275 10621 17828 0 15649 1567 63921 0 626 0 608 (radio 1.35% / 0.95% tx 0.50% / 0.00% listen 0.85% / 0.95%)     |
| 01:05.234 | ID:3  | 8198 P 0.18 31 62560 2034487 8006 18742 0 15154 1789 63699 0 737 0 378 (radio 1.27% / 1.12% tx 0.38% / 0.00% listen 0.89% / 1.12%)      |

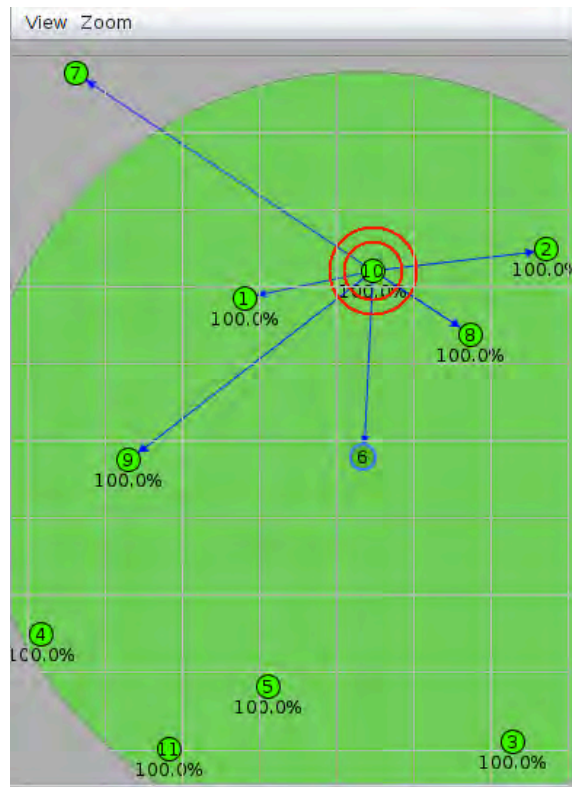
Figure 59: Mote output broadcast power trace data

The displayed parameters are the following:

**Table 6: PowerTrace Output Parameters**

| Parameter and value    | Explanation   |
|------------------------|---|
| clock_time() = 2408    | clock time  |
| rimeaddr = 193, 250    | rime address  |
| seqno = 17             | sequence number                                     |
| all_cpu = 11309        | accumulated CPU energy consumption                  |
| all_lpm = 578542       | accumulated Low Power Mode energy consumption       |
| all_transmit = 0       | accumulated transmission energy consumption         |
| all_listen = 2574      | accumulated listen energy consumption               |
| all_idle_transmit = 0  | accumulated idle transmission energy consumption    |
| all_idle_listen = 2574 | accumulated idle listen energy consumption          |
| cpu = 689              | CPU energy consumption for this cycle               |
| lpm = 32078            | LPM energy consumption for this cycle               |
| transmit = 0           | transmission energy consumption for this cycle      |
| listen = 144           | listen energy consumption for this cycle            |
| idle_transmit = 0      | idle transmission energy consumption for this cycle |
| idle_listen = 144      | idle listen energy consumption for this cycle       |

The powerTrace consumption of this scenario is saved into a file called: `loglistener.txt` and it will be compared to the same execution enabling cryptography on the motes. Before re executing the same scenario enabling cryptography between nodes another run is set to run with Z1 motes. The scenario is set again as an UDP-RPL broadcast communication environment using the following network structure:



**Figure 60: Broadcast WSN, Z1 mote**

The powerTrace consumption of this scenario is also saved into a file called: `loglistener2.txt` and it will be compared to the same execution enabling cryptography on the motes.

It is important to note that, during all simulations (including this WSN **broadcast** scenario) we place the motes in the same physical distribution **maintaining the same distance magnitude order** than the first two cases, also the mote number tags remain the same that in simulations 3 and 4.



## 4.6. Simulations 5/6 – Sky mote, Z1 mote – Broadcast WSN simulation encryption

As explained in section 4.5, the `project_conf.h` file will be modified to enable data encryption between nodes.

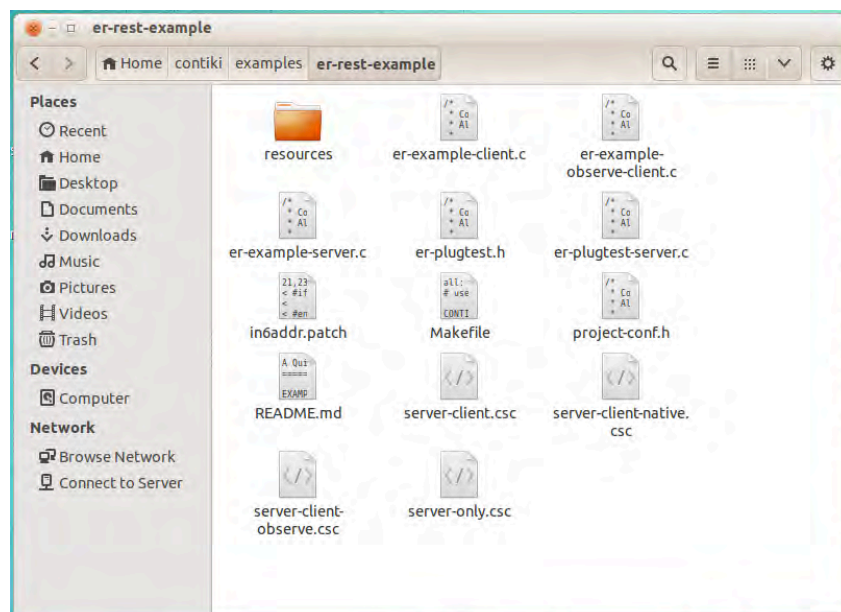


Figure 61: `project_conf.h` file location

A backup of the file is generated (as it is part of the O.S system and it must be kept intact in case of any malfunctioning). The following lines are added to the file; in our case the set encryption level is the following:

**0x04** AES-CTR ENC Data is encrypted. Data authenticity is not validated.

```

/* Enable message encryption */
#define AES_128_conf aes_128_driver
#undef LLSEC802154_CONF_ENABLED
#define LLSEC802154_CONF_ENABLED 1
#undef NETSTACK_CONF_FRAMER
#define NETSTACK_CONF_FRAMER noncoresec_framer
#undef NETSTACK_CONF_LLSEC
#define NETSTACK_CONF_LLSEC noncoresec_driver
#undef NONCORESEC_CONF_SEC_LVL
#define NONCORESEC_CONF_SEC_LVL 0x04

```

root@instant-contiki: /home/user/contiki/tools/coolja

Figure 62: `project_conf.h` file modification

Once encryption is enabled a new broadcast run is generated using the same base functions and firmware: `broadcast-example-2.c` and `broadcast-example-2.sky`. As it can be seen in this first run the used motes are sky motes, the message transmission is performed with data encrypted but data authenticity is not validated. Also encryption is performed under the AES\_128 standard driver.

As it was commented, the generated scenario is the same:

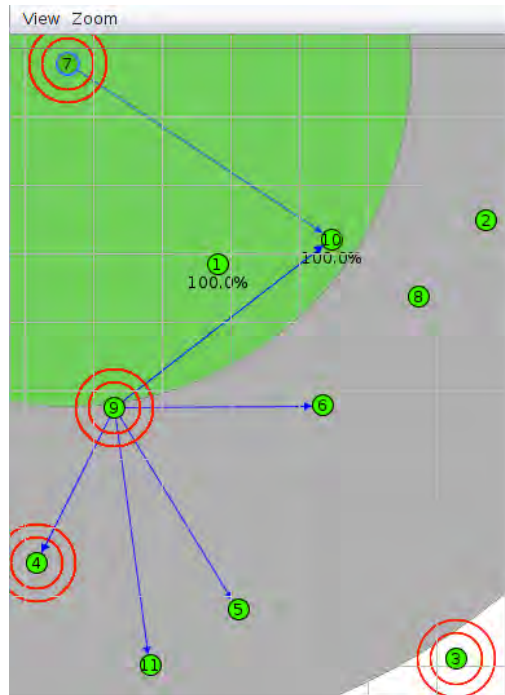


Figure 63: Broadcast WSN, Sky mote, encryption enabled.

And once again, the output data is stored in the corresponding loglistener file, ready to be compared with the other simulation runs.

As stated before, another simulation run is generated, but this time using Z1 mote types.

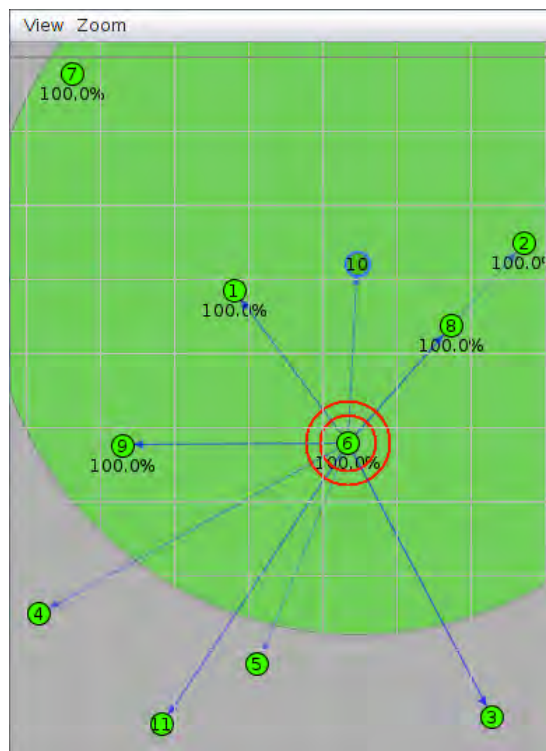


Figure 64: Broadcast WSN, Z1 mote, encryption enabled.

At this point, the following log files are available to be compared:

- Broadcast scenario, Sky motes. No encryption.
- Broadcast scenario, Z1 motes. No encryption.
- Broadcast scenario. Sky motes. Encryption.
- Broadcast scenario. Z1 motes. Encryption

In the Analysis of the Simulation Results section we will work through these logs. Before these comparisons, two extra scenarios will be simulated using the software-based AES driver enabling encryption just like it has been done with simulations 5 and 6.

In this following case, the simulation will be performed using the first scenarios (showed in sections 4.2 and 4.3), taking advantage of the Sensor Data Collector tool. This tool will also allow us to compare the first two scenarios with the newly generated ones.

Once again, note how the two scenarios have the same mote placement it only differs on the mote tab number between simulation 5 and 6. (Which is the same tag number distribution that in simulations 3 and 4)

### 4.7. Simulation 7 – Sky mote – Sink-Sender – WSN with cryptography allowed

For this case the same network distribution (Sink-sender) than simulations 1 and 2 is used:

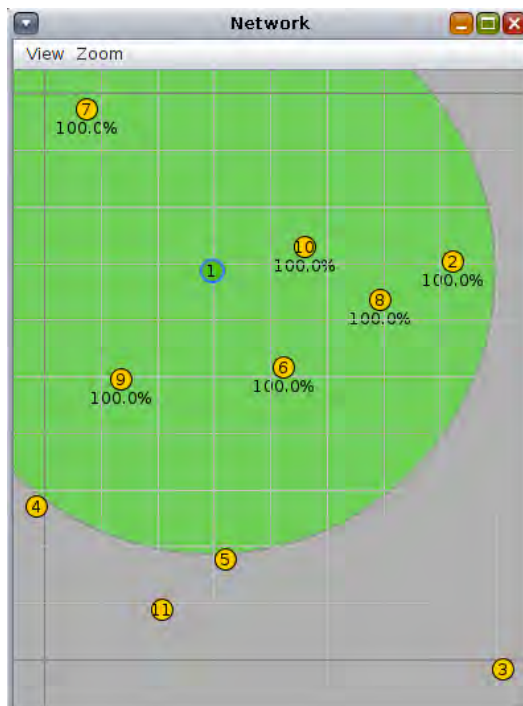


Figure 65: Sky motes, Sink-Sender scenario with cryptography

As it can be seen in the upper figure the Sink module is represented with the green circle whereas the sender motes are represented using the yellow colour. After the scenario is set, the Sensor Data Collector tool is initialized and the simulation run starts.

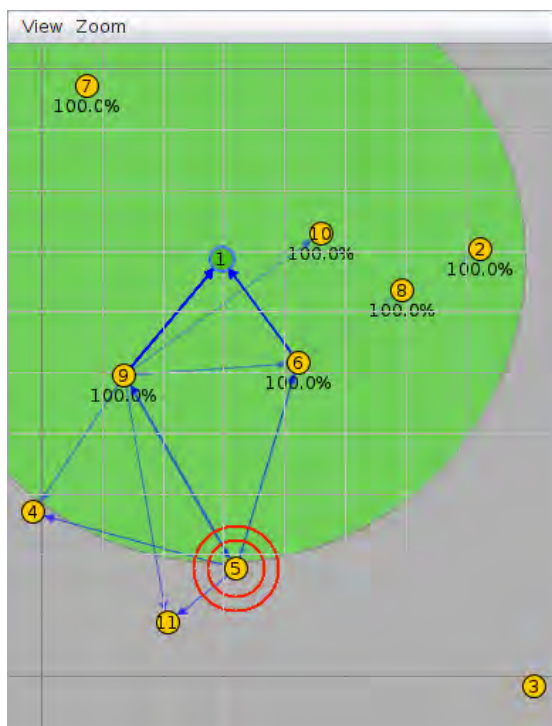


Figure 66: Sink-Sender simulation going on



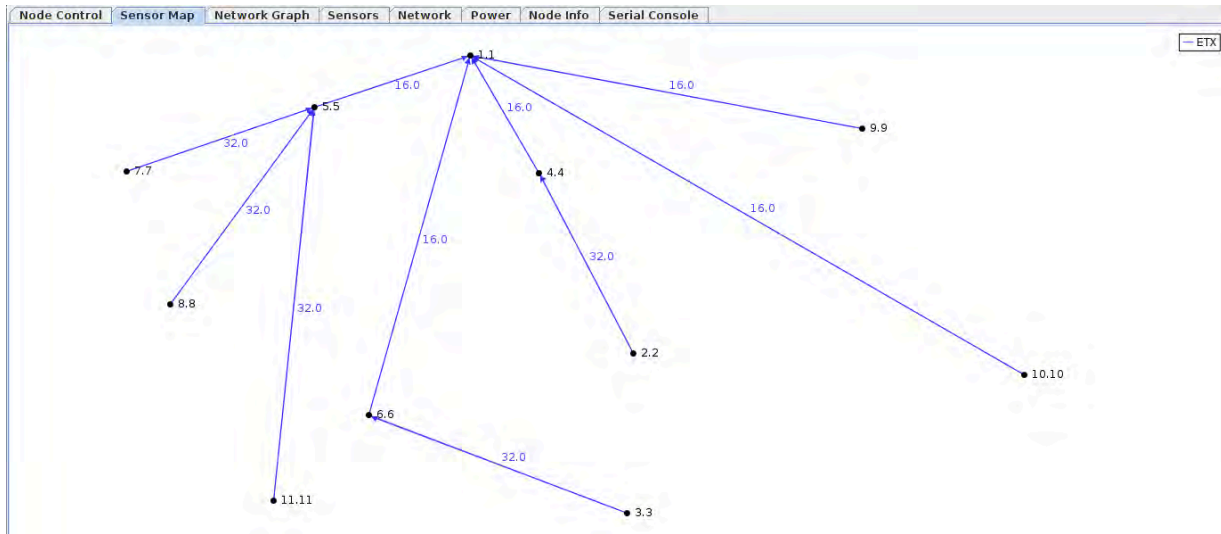


Figure 67: Sensor Map

| Node Control |       | Sensor Map |        | Network Graph |                 | Sensors |           | Network   |              | Power          |       | Node Info     |                   | Serial Console      |                       |
|--------------|-------|------------|--------|---------------|-----------------|---------|-----------|-----------|--------------|----------------|-------|---------------|-------------------|---------------------|-----------------------|
| Lost         | Hops  | Rtmetric   | ETX    | Churn         | Beacon Interval | Reboots | CPU Power | LPM Power | Listen Power | Transmit Power | Power | On-time       | Listen Duty Cycle | Transmit Duty Cycle | Avg Inter-packet Time |
| 0            | 0.000 | 0.000      | 0.000  | 0             |                 | 0       | 0.000     | 0.000     | 0.000        | 0.000          | 0.000 |               | 0.000             | 0.000               |                       |
| 0            | 2.000 | 768.000    | 32.000 | 0             | 3 min, 38 sec   | 0       | 0.401     | 0.151     | 0.510        | 0.247          | 1.309 | 0 min, 37 sec | 0.850             | 0.465               | 0 min, 27 sec         |
| 0            | 2.000 | 808.500    | 32.000 | 0             | 2 min, 11 sec   | 0       | 0.385     | 0.152     | 0.560        | 0.379          | 1.476 | 0 min, 28 sec | 0.934             | 0.713               | 0 min, 07 sec         |
| 0            | 1.000 | 512.000    | 16.000 | 0             | 2 min, 11 sec   | 0       | 0.505     | 0.148     | 0.614        | 0.400          | 1.667 | 0 min, 29 sec | 1.024             | 0.753               | 0 min, 08 sec         |
| 0            | 1.000 | 512.000    | 16.000 | 0             | 3 min, 16 sec   | 0       | 0.510     | 0.148     | 0.631        | 0.231          | 1.519 | 0 min, 23 sec | 1.052             | 0.434               | 0 min, 19 sec         |
| 0            | 1.000 | 512.000    | 16.000 | 0             | 3 min, 38 sec   | 0       | 0.454     | 0.150     | 0.545        | 0.179          | 1.328 | 0 min, 27 sec | 0.909             | 0.337               | 0 min, 35 sec         |
| 0            | 2.000 | 804.000    | 32.000 | 0             | 3 min, 38 sec   | 0       | 0.434     | 0.150     | 0.537        | 0.213          | 1.335 | 0 min, 39 sec | 0.896             | 0.401               | 0 min, 36 sec         |
| 0            | 2.000 | 768.000    | 32.000 | 0             | 3 min, 38 sec   | 0       | 0.409     | 0.151     | 0.508        | 0.187          | 1.255 | 0 min, 31 sec | 0.847             | 0.352               | 0 min, 40 sec         |
| 0            | 1.000 | 512.000    | 16.000 | 0             | 3 min, 38 sec   | 0       | 0.443     | 0.150     | 0.498        | 0.103          | 1.194 | 0 min, 37 sec | 0.830             | 0.194               | 0 min, 42 sec         |
| 0            | 1.000 | 512.000    | 16.000 | 0             | 3 min, 16 sec   | 0       | 0.359     | 0.153     | 0.450        | 0.184          | 1.145 | 0 min, 24 sec | 0.750             | 0.346               | 0 min, 23 sec         |
| 0            | 2.000 | 768.000    | 32.000 | 0             | 3 min, 16 sec   | 0       | 0.344     | 0.153     | 0.465        | 0.254          | 1.216 | 0 min, 19 sec | 0.775             | 0.478               | 0 min, 40 sec         |
| 0.000        | 1.500 | 647.650    | 24.000 | 0.000         | 3 min, 14 sec   | 0.000   | 0.424     | 0.151     | 0.532        | 0.238          | 1.344 | 0 min, 29 sec | 0.887             | 0.447               | 0 min, 28 sec         |

Figure 68: Node Info table (I)

| Min Inter-packet Time | Max Inter-packet Time |
|-----------------------|-----------------------|
|                       |                       |
| 0 min, 18 sec         | 1 min, 05 sec         |
| 0 min, 14 sec         | 0 min, 14 sec         |
| 0 min, 17 sec         | 0 min, 17 sec         |
| 0 min, 39 sec         | 0 min, 39 sec         |
| 0 min, 37 sec         | 1 min, 10 sec         |
| 0 min, 48 sec         | 1 min, 01 sec         |
| 0 min, 47 sec         | 1 min, 13 sec         |
| 0 min, 30 sec         | 1 min, 36 sec         |
| 0 min, 47 sec         | 0 min, 47 sec         |
| 1 min, 21 sec         | 1 min, 21 sec         |
| <b>0 min, 37 sec</b>  | <b>0 min, 56 sec</b>  |

Figure 69: Node Info table (II)

As we did with the two previous simulations performed in sections 4.2 and 4.3 this table is highly useful to compare the performance between the various scenarios. Again, it can be seen how the average power consumed by the motes is **1'344mW** (a sum of CPU power, LPM Power, Listen Power and Transmit Power), while in the non-encryption case was **1'138mW**. The average Inter-packet Time is 28 seconds and the average Transmit Duty Cycle, the fraction of one period in which a the system is active, is **0'447**. It also shows information about the average Beacon interval, one of the management frames in IEEE 802.11 based WLANs. It contains all the information about the network. Beacon frames are transmitted periodically, they serve to

announce the presence of a wireless LAN and to synchronise the members of the service set, in this case it is 3min and 14sec while in the first Sky motes scenario it was 5 min and 15sec.

Power distribution on each mote as well as Radio Duty Cycle figures are shown next:

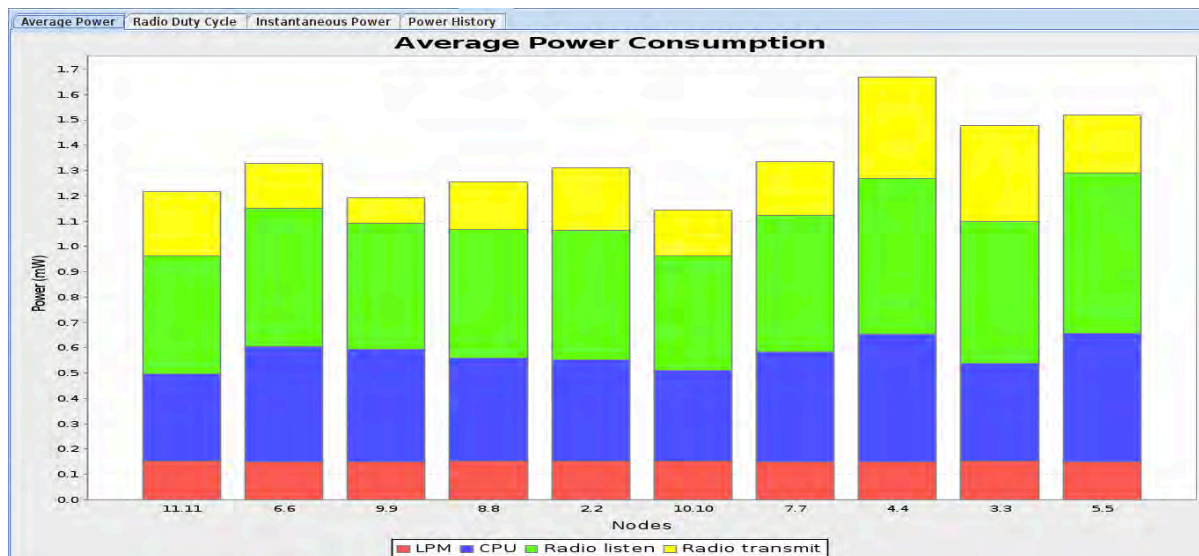


Figure 70: Average Power Consumption

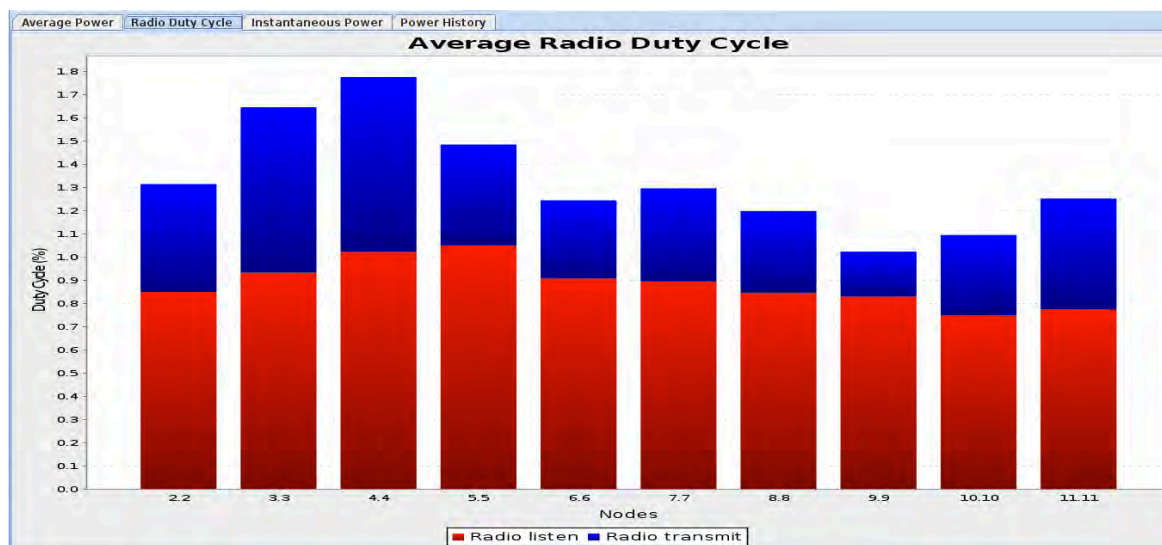


Figure 71: Average Radio Duty Cycle

The upper figure shows in a visual way that the average Transmit Duty Cycle is higher than the case when no encryption is enabled. Indeed it is almost the double 0'447 vs 0'241. This fact shows that due to the message encryption the period of time in which the system is active is higher than the case where no encryption is enabled. In fact it is **92'73% higher**. Also, regarding the average power consumption it is also higher in the encryption case.

Again, we will go through more details in the Analysis of the simulation results section.

### 4.8. Simulation 8 – Z1 mote – Sink-Sender – WSN with cryptography allowed

For this scenario the same network distribution is that case 4.7 is recreated

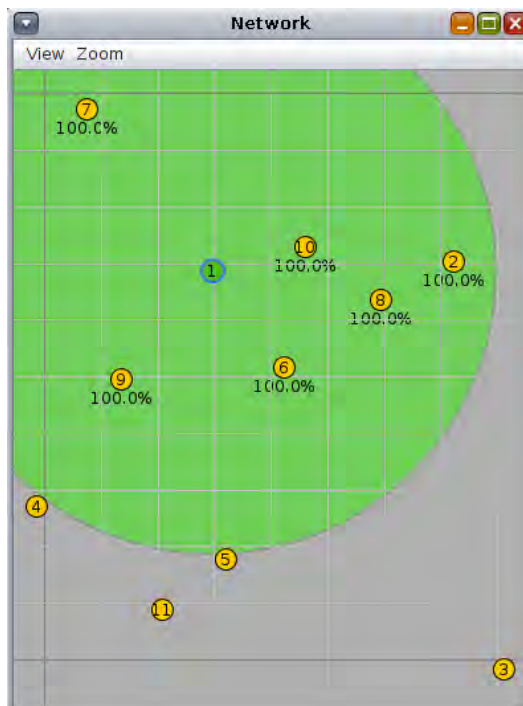


Figure 72: Z1 motes, Sink-Sender scenario with cryptography

Just like we did in the previous scenario a sink mote and 10 sender motes are placed into the environment. After that the Data Sensor Collector Tool is activated so the information about the motes performance can be captured.

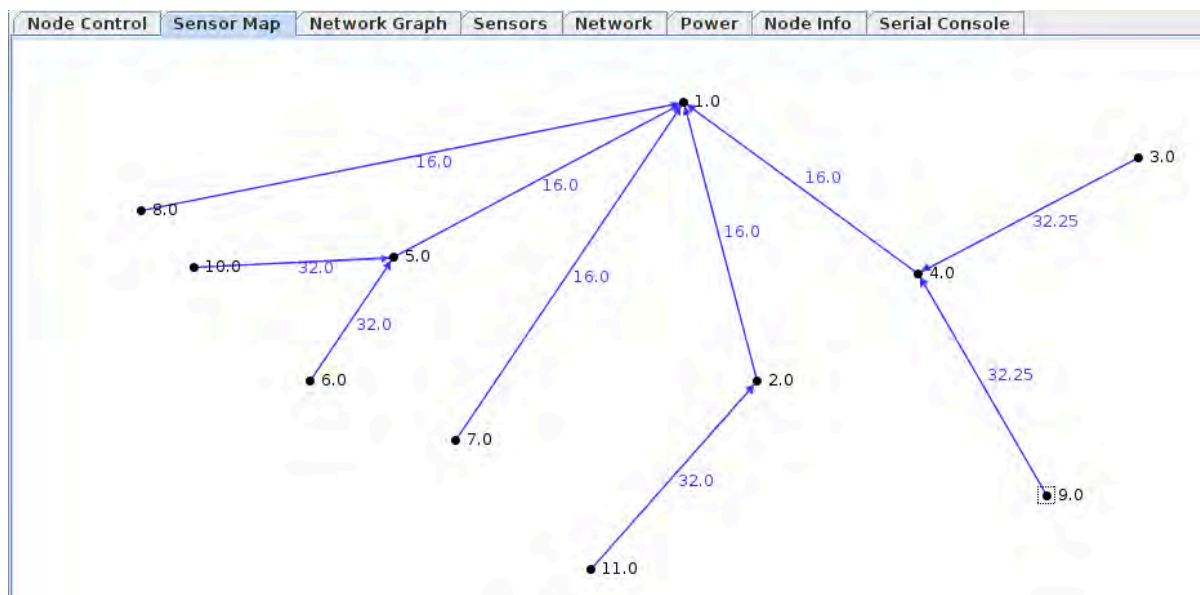


Figure 73: Z1 motes, sensor map

| Nodes<br><All> | Node Control |              | Sensor Map   |              | Network Graph  |               | Sensors      | Network              | Power           | Node Info    |              | Serial Console |              |                |                      |             |                  |
|----------------|--------------|--------------|--------------|--------------|----------------|---------------|--------------|----------------------|-----------------|--------------|--------------|----------------|--------------|----------------|----------------------|-------------|------------------|
|                | Node         | Received     | Dups         | Lost         | Hops           | Rtmetric      | ETX          | Churn                | Beacon Interval | Reboots      | CPU Power    | LPM Power      | Listen Power | Transmit Power | Power                | On-time     | Listen Duty Cycl |
| 1.0            | 0            | 0            | 0            | 0.000        | 0.000          | 0.000         | 0            |                      | 0               | 0.000        | 0.000        | 0.000          | 0.000        | 0.000          | 0.000                |             | 0.00             |
| 2.0            | 2            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 3 min, 16 sec        | 0               | 0.161        | 0.159        | 0.332          | 0.214        | 0.865          | 0 min, 27 sec        | 0.55        |                  |
| 3.0            | 2            | 0            | 0            | 2.000        | 788.000        | 32.625        | 0            | 1 min, 38 sec        | 0               | 0.090        | 0.161        | 0.267          | 0.343        | 0.861          | 0 min, 23 sec        | 0.44        |                  |
| 4.0            | 2            | 0            | 0            | 1.000        | 516.000        | 16.000        | 0            | 2 min, 11 sec        | 0               | 0.125        | 0.160        | 0.283          | 0.223        | 0.791          | 0 min, 19 sec        | 0.47        |                  |
| 5.0            | 2            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 2 min, 11 sec        | 0               | 0.162        | 0.159        | 0.299          | 0.241        | 0.861          | 0 min, 20 sec        | 0.49        |                  |
| 6.0            | 2            | 0            | 0            | 2.000        | 768.000        | 32.000        | 0            | 3 min, 16 sec        | 0               | 0.098        | 0.161        | 0.273          | 0.230        | 0.762          | 0 min, 24 sec        | 0.45        |                  |
| 7.0            | 2            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 3 min, 16 sec        | 0               | 0.110        | 0.160        | 0.260          | 0.167        | 0.696          | 0 min, 25 sec        | 0.43        |                  |
| 8.0            | 2            | 0            | 0            | 1.000        | 512.000        | 16.000        | 0            | 2 min, 11 sec        | 0               | 0.090        | 0.161        | 0.218          | 0.173        | 0.642          | 0 min, 22 sec        | 0.36        |                  |
| 9.0            | 2            | 0            | 0            | 2.000        | 794.000        | 32.375        | 0            | 3 min, 16 sec        | 0               | 0.117        | 0.160        | 0.333          | 0.276        | 0.887          | 0 min, 24 sec        | 0.55        |                  |
| 10.0           | 2            | 0            | 0            | 2.000        | 768.000        | 32.000        | 0            | 3 min, 16 sec        | 0               | 0.101        | 0.160        | 0.287          | 0.301        | 0.849          | 0 min, 17 sec        | 0.47        |                  |
| 11.0           | 1            | 0            | 0            | 2.000        | 768.000        | 32.000        | 0            | 2 min, 11 sec        | 0               | 0.113        | 0.160        | 0.327          | 0.379        | 0.979          | 0 min, 09 sec        | 0.54        |                  |
| <b>Avg</b>     | <b>1.900</b> | <b>0.000</b> | <b>0.000</b> | <b>1.500</b> | <b>645.000</b> | <b>24.100</b> | <b>0.000</b> | <b>2 min, 40 sec</b> | <b>0.000</b>    | <b>0.117</b> | <b>0.160</b> | <b>0.288</b>   | <b>0.255</b> | <b>0.819</b>   | <b>0 min, 21 sec</b> | <b>0.48</b> |                  |

Figure 74: Node Info table (I)

| Transmit Duty Cycle | Avg Inter-packet Time | Min Inter-packet Time | Max Inter-packet Time |
|---------------------|-----------------------|-----------------------|-----------------------|
| 0.000               |                       |                       |                       |
| 0.403               | 0 min, 13 sec         | 0 min, 27 sec         | 0 min, 27 sec         |
| 0.646               | 0 min, 33 sec         | 1 min, 07 sec         | 1 min, 07 sec         |
| 0.420               | 0 min, 19 sec         | 0 min, 38 sec         | 0 min, 38 sec         |
| 0.454               | 0 min, 19 sec         | 0 min, 39 sec         | 0 min, 39 sec         |
| 0.434               | 0 min, 27 sec         | 0 min, 55 sec         | 0 min, 55 sec         |
| 0.314               | 0 min, 30 sec         | 1 min, 01 sec         | 1 min, 01 sec         |
| 0.326               | 0 min, 23 sec         | 0 min, 47 sec         | 0 min, 47 sec         |
| 0.520               | 0 min, 28 sec         | 0 min, 56 sec         | 0 min, 56 sec         |
| 0.567               | 0 min, 32 sec         | 1 min, 05 sec         | 1 min, 05 sec         |
| 0.714               |                       |                       |                       |
| <b>0.480</b>        | <b>0 min, 22 sec</b>  | <b>0 min, 45 sec</b>  | <b>0 min, 45 sec</b>  |

Figure 75: Node Info table (II)

One more time, following the schema used in the previous section, this table is highly useful to compare the performance between these two scenarios, Sky motes with and without cryptography and Z1 motes with and without cryptography enabled. In this case, it can be seen how the average power consumed by the motes is **0’819mW** (a sum of CPU power, LPM Power, Listen Power and Transmit Power). This value is lower than the Sky motes case although it exists in the same magnitude order. When we compare it with the case in which the Z1 motes were not encrypting the messages it can be seen how the average consumed power is **increased by a 67’57%**, from 0’606mW to 0’819mW. In this case, the average Inter-packet Time is 22 seconds and the average Transmit Duty Cycle, the fraction of one period in which a the system is active, is **0’480**, this value is also similar than the Sky motes case, where it was 0’447, but in the Z1 motes scenario where cryptography was not enabled this value was 0’213. This means that the average Transmit Duty Cycle is **increased by 112’67%**. As usual we also have information about the average Beacon interval, in this case it is 2min and 40sec while in the first Z1 motes scenario it was 5 min and 19sec.

Power distribution on each mote as well as Radio Duty Cycle figures are shown next:



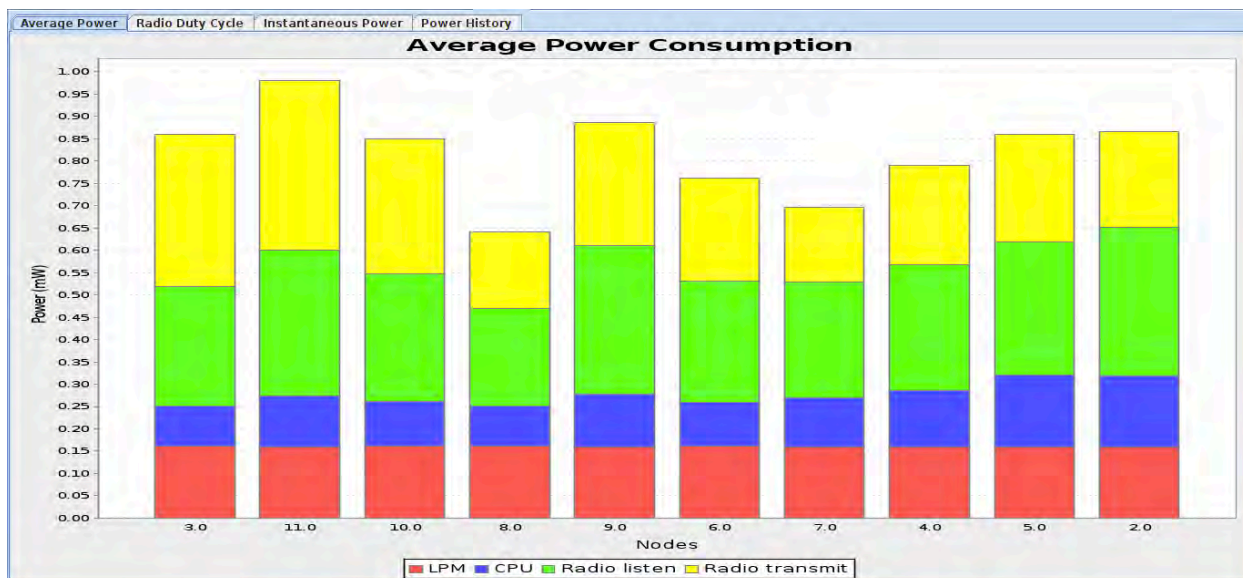


Figure 76: Average Power Consumption

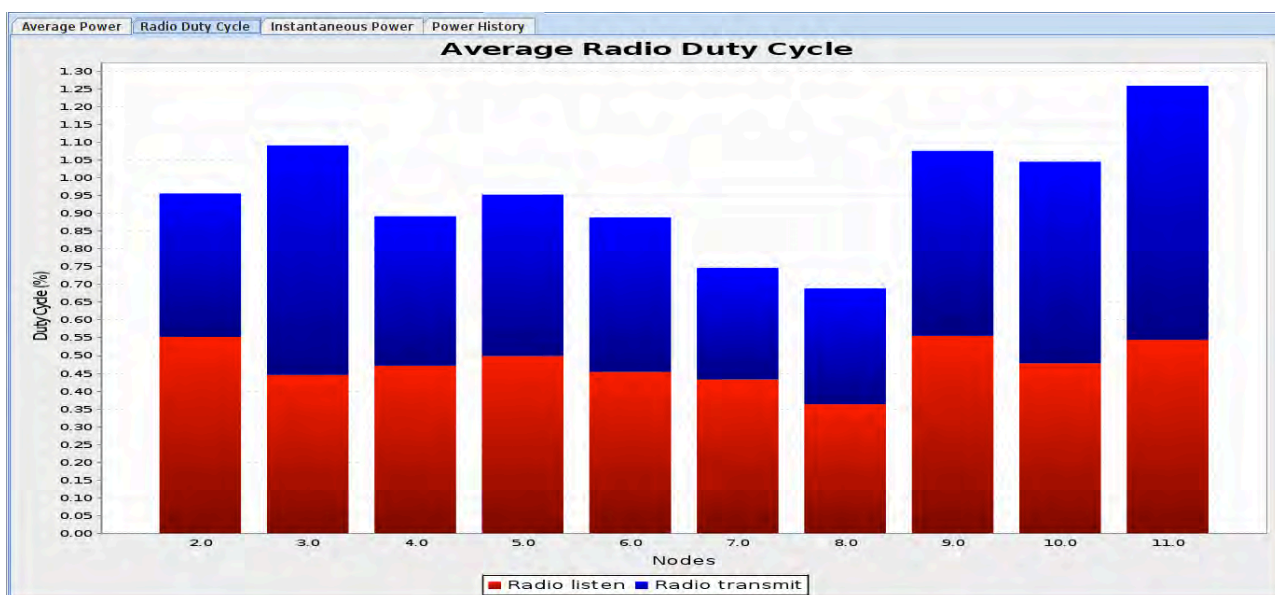


Figure 77: Average Radio Duty Cycle

## 4.9. Analysis of the Simulation Results

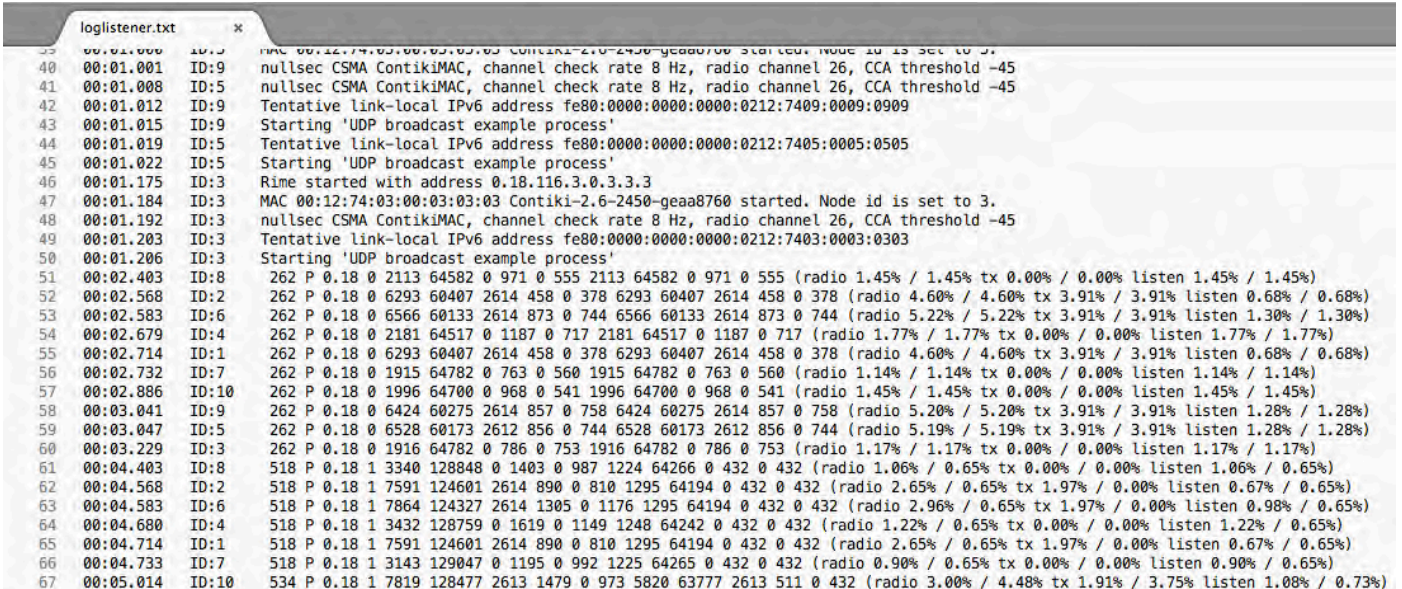
To sum up, the following simulations have been performed:

1. Sink-sender environment: Sky motes. No encryption.
2. Sink-sender environment: Z1 motes. No encryption.
3. Broadcast scenario: Sky motes. No encryption.
4. Broadcast scenario: Z1 motes. No encryption.
5. Broadcast scenario: Sky motes. Encryption.
6. Broadcast scenario: Z1 motes. Encryption.
7. Sink-sender environment: Sky motes. Encryption.
8. Sink-sender environment: Z1 motes. Encryption

As commented in the previous sections (4.5 – 4.6) a log file is generated for the broadcast scenario simulations (3 to 6). This is useful in order to compare performance parameters between these simulation sets with and without encryption. We will first work with these four scenarios and then we will move back to the other simulation sets, the sink-sender environment.

### 4.9.1. Broadcast scenario

As stated before, the log files are called `loglistener.txt` (0-3). These files were generated by the Cooja simulator after adding the `powertrace` functionality explained in section 4.5. They contain the information listed in **Table 6** (section 4.5).



```

loglistener.txt
50 00:01:000 ID:3 MAC 00:12:74:03:00:03:03:03 Contiki-2.6-2450-geaa8760 started. Node id is set to 3.
40 00:01:001 ID:9 nullsec CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26, CCA threshold -45
41 00:01:008 ID:5 nullsec CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26, CCA threshold -45
42 00:01:012 ID:9 Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7409:0009:0909
43 00:01:015 ID:9 Starting 'UDP broadcast example process'
44 00:01:019 ID:5 Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7405:0005:0505
45 00:01:022 ID:5 Starting 'UDP broadcast example process'
46 00:01:175 ID:3 Rime started with address 0.18.116.3.0.3.3.3
47 00:01:184 ID:3 MAC 00:12:74:03:00:03:03:03 Contiki-2.6-2450-geaa8760 started. Node id is set to 3.
48 00:01:192 ID:3 nullsec CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26, CCA threshold -45
49 00:01:203 ID:3 Tentative link-local IPv6 address fe80:0000:0000:0000:0212:7403:0003:0303
50 00:01:206 ID:3 Starting 'UDP broadcast example process'
51 00:02:403 ID:8 262 P 0.18 0 2113 64582 0 971 0 555 2113 64582 0 971 0 555 (radio 1.45% / 1.45% tx 0.00% / 0.00% listen 1.45% / 1.45%)
52 00:02:568 ID:2 262 P 0.18 0 6293 60407 2614 458 0 378 6293 60407 2614 458 0 378 (radio 4.60% / 4.60% tx 3.91% / 3.91% listen 0.68% / 0.68%)
53 00:02:583 ID:6 262 P 0.18 0 6566 60133 2614 873 0 744 6566 60133 2614 873 0 744 (radio 5.22% / 5.22% tx 3.91% / 3.91% listen 1.30% / 1.30%)
54 00:02:679 ID:4 262 P 0.18 0 2181 64517 0 1187 0 717 2181 64517 0 1187 0 717 (radio 1.77% / 1.77% tx 0.00% / 0.00% listen 1.77% / 1.77%)
55 00:02:714 ID:1 262 P 0.18 0 6293 60407 2614 458 0 378 6293 60407 2614 458 0 378 (radio 4.60% / 4.60% tx 3.91% / 3.91% listen 0.68% / 0.68%)
56 00:02:732 ID:7 262 P 0.18 0 1915 64782 0 763 0 560 1915 64782 0 763 0 560 (radio 1.14% / 1.14% tx 0.00% / 0.00% listen 1.14% / 1.14%)
57 00:02:886 ID:10 262 P 0.18 0 1996 64700 0 968 0 541 1996 64700 0 968 0 541 (radio 1.45% / 1.45% tx 0.00% / 0.00% listen 1.45% / 1.45%)
58 00:03:041 ID:9 262 P 0.18 0 6424 60275 2614 857 0 758 6424 60275 2614 857 0 758 (radio 5.20% / 5.20% tx 3.91% / 3.91% listen 1.28% / 1.28%)
59 00:03:047 ID:5 262 P 0.18 0 6528 60173 2612 856 0 744 6528 60173 2612 856 0 744 (radio 5.19% / 5.19% tx 3.91% / 3.91% listen 1.28% / 1.28%)
60 00:03:229 ID:3 262 P 0.18 0 1916 64782 0 786 0 753 1916 64782 0 786 0 753 (radio 1.17% / 1.17% tx 0.00% / 0.00% listen 1.17% / 1.17%)
61 00:04:403 ID:8 518 P 0.18 1 3340 128848 0 1403 0 987 1224 64266 0 432 0 432 (radio 1.06% / 0.65% tx 0.00% / 0.00% listen 1.06% / 0.65%)
62 00:04:568 ID:2 518 P 0.18 1 7591 124601 2614 890 0 810 1295 64194 0 432 0 432 (radio 2.65% / 0.65% tx 1.97% / 0.00% listen 0.67% / 0.65%)
63 00:04:583 ID:6 518 P 0.18 1 7864 124327 2614 1305 0 1176 1295 64194 0 432 0 432 (radio 2.96% / 0.65% tx 1.97% / 0.00% listen 0.98% / 0.65%)
64 00:04:680 ID:4 518 P 0.18 1 3432 128759 0 1619 0 1149 1248 64242 0 432 0 432 (radio 1.22% / 0.65% tx 0.00% / 0.00% listen 1.22% / 0.65%)
65 00:04:714 ID:1 518 P 0.18 1 7591 124601 2614 890 0 810 1295 64194 0 432 0 432 (radio 2.65% / 0.65% tx 1.97% / 0.00% listen 0.67% / 0.65%)
66 00:04:733 ID:7 518 P 0.18 1 3143 129047 0 1195 0 992 1225 64265 0 432 0 432 (radio 0.90% / 0.65% tx 0.00% / 0.00% listen 0.90% / 0.65%)
67 00:05:014 ID:10 534 P 0.18 1 7819 128477 2613 1479 0 973 5820 63777 2613 511 0 432 (radio 3.00% / 4.48% tx 1.91% / 3.75% listen 1.08% / 0.73%)

```

Figure 78: loglistener.txt file portion

As shown in the upper figure, those four files have a plain text format. After extracting all the data we are able to compare the following features:

- Accumulated CPU energy consumption.
- Accumulated Low Power Mode energy consumption.
- Accumulated Transmission energy consumption.
- Accumulated Listen energy consumption.
- Accumulated idle Transmission energy consumption.
- Accumulated idle Listen energy consumption.

From the four simulations, the extracted data is shown next:

**a. Sky motes. No encryption:**

- Accumulated CPU energy consumption: 146.836mW
- Accumulated Low Power Mode energy consumption: 4.242.493mW
- Accumulated Transmission energy consumption: 24.018mW
- Accumulated Listen energy consumption: 39.961mW
- Accumulated idle Transmission energy consumption: 0mW
- Accumulated idle Listen energy consumption: 31.621mW

**b. Z1 motes. No encryption:**

- Accumulated CPU energy consumption: 100.240mW
- Accumulated Low Power Mode energy consumption: 4.290.566mW
- Accumulated Transmission energy consumption: 21.387mW
- Accumulated Listen energy consumption: 20.477mW
- Accumulated idle Transmission energy consumption: 0mW
- Accumulated idle Listen energy consumption: 0mW

**c. Sky motes. Encryption:**

- Accumulated CPU energy consumption: 148.330mW
- Accumulated Low Power Mode energy consumption: 4.241.020mW
- Accumulated Transmission energy consumption: 24.010mW
- Accumulated Listen energy consumption: 40.501mW
- Accumulated idle Transmission energy consumption: 0mW
- Accumulated idle Listen energy consumption: 29.876mW

**d. Z1 motes. Encryption:**

- Accumulated CPU energy consumption: 95.208mW
- Accumulated Low Power Mode energy consumption: 4.295.600mW
- Accumulated Transmission energy consumption: 18.984mW
- Accumulated Listen energy consumption: 18.818mW
- Accumulated idle Transmission energy consumption: 0mW
- Accumulated idle Listen energy consumption: 0mW

*\*All the data was extracted at the same sequence number: 66 from the central mote in each of the broadcast scenarios.*

From all these data we can see the following points:

- In a broadcast scenario with simple data exchange, like the one presented in these simulation sets, the application of message ciphering (cryptography) does not affect the system performance at all.
- The two mote types under study (Z1 and Sky) behave in a similar way under the same circumstances and a similar (almost equal) mote distribution. The application of the software-based AES driver of 128 bits. It has a fixed block size of 128 bits and a 128 bit key size.
- The performance behaviour proves that Block Ciphers are applicable in IoT environments with small and power constrained devices such as the Sky motes and the Zolertia Z1 motes. As it will also be commented on the next section, this is highly dependable on the application needs. As more data is to be managed by the system and more complex operations need to be performed by the devices, this becomes performance starts to downgrade and the cryptography application becomes critical. For our simulation case, motes exchanging simple data to the others, this is not a crucial aspect, as the performance is not even affected. This could be the case of sensors obtaining data from the environment and transmitting it to the others, generating an information map.
- The energy consumption levels of both Sky and Z1 motes are acceptable as part of Low Power device standards in both cases, No use of cryptography and cryptography enabled.
- Let's remember that we are dealing in a "non-authenticated data" set. If information is to be validated, then this performance downgrades. This raises an important and recurrent topic discussed in this project; the demanded security levels for IoT applications must be well though due to devices performance constraints. Nevertheless these security levels are to be set specifically depending on each application needs and cannot be generalized for every IoT application.

Now, we can go back to the other four simulation sets; the sink-sender environment.



#### 4.9.2. Sink-Sender environment

To analyse these simulations we take into account the following figures:

##### Data Analysis Figures:

1. Sink-sender environment: Sky motes. No encryption.
  - a. Figure 44
  - b. Figure 45
2. Sink-sender environment: Z1 motes. No encryption.
  - a. Figure 52
  - b. Figure 53
3. Sink-sender environment: Sky motes. Encryption.
  - a. Figure 68
  - b. Figure 69
4. Sink-sender environment: Z1 motes. Encryption.
  - a. Figure 74
  - b. Figure 75

##### Power Analysis Figures:

5. Sink-sender environment: Sky motes. No encryption.
  - a. Figure 46
  - b. Figure 47
6. Sink-sender environment: Z1 motes. No encryption.
  - a. Figure 54
  - b. Figure 55
7. Sink-sender environment: Sky motes. Encryption.
  - a. Figure 70
  - b. Figure 71
8. Sink-sender environment: Z1 motes. Encryption.
  - a. Figure 76
  - b. Figure 77

##### Sky mote case

From all the data extracted from the simulation run, it can be seen how the average power consumed by the motes is **1'344mW** (a sum of CPU power, LPM Power, Listen Power and Transmit Power), while in the non-encryption case was **1'138mW**. While not a significant increase, this fact proves that in an application like this, in which not only message exchange is performed, but also system synchronization, the power consumption suffered by the motes increases in the encryption case, due to the ciphering process forced by the AES 128 standard.

Also, the Transmit Duty Cycle for the first case is **0'241**, while in the encryption case it goes up to **0'447**. The more operations to perform (encryption) and data to transmit the higher the Transmit Duty Cycle is going to be. This is no secret and totally expected, the system is busier under the application of cryptography protocols, once again the topic is raised: *The expected security levels for IoT applications must be well though due to devices performance constraints*. We see how, in this case, the system is, on average, two times busier when the AES standard is applied.

Regarding the Beacon interval, while the non-encryption case showed a total of 5 min and 15sec, in the second case it goes down to 3min and 14sec. This is also interesting, as Beacons are needed for the devices or clients to receive information about the particular sink module (or, for example, a router) a Beacon would include some main information such as SSID, Timestamp, and various parameters. Beacons broadcasted by the sink mote take up some of the bandwidth that can be used for the actual data transmission. So by having higher Beacon interval values, better throughput and better speed will be achieved, leading up to better overall performance. The connected devices will also have better battery life, as the wireless adapter card is able to “sleep” in between the beacon broadcasts, the devices save energy consumption which equate to longer battery life. Basically what we see here is that in the encryption case more beacons are needed in order to correctly synchronize the network (as the Beacon interval is lower) thus leading to more energy consumption. This is leading into the average power consumption figure, higher in the encryption case.

### **Z1 mote case**

The exact same analysis is applied into the Z1 motes case, with the lonely difference appearing in the parameter values.

In this case, the average power consumed by the motes is **0'606mW** when no encryption is applied and **0'819mW** in the encryption case, always lower than the Sky motes cases although in the same magnitude order. The average Transmit Duty Cycle when encryption is enabled is **0'480**, also similar than the Sky motes case. When cryptography was not enabled this value was **0'213**.

Finally, in this case the Beacon interval time is 2min and 40sec while in the first Z1 motes scenario it was 5 min and 19sec. Once again, proving that in the encryption case more beacons are needed in order to correctly synchronize the network (as the Beacon interval is lower) thus leading to more energy consumption.

## 5. Results and Conclusions

This “Results and Conclusions” section will have the following order; first the results of the block and stream ciphers studied will be presented, adding the conclusions extracted from those studies and also answering some of the raised questions of those sections.

Then we will jump to the SHA-3 algorithm and Hash study section, where we will also talk about the conclusion observed from comparing this structure to the firstly presented ciphers. We will also talk about how the simulations helped understand the hashing process.

To end this chapter we will see the conclusions obtained from the study of the Elliptic Curve Cryptography and its current applications.

Finally, we will link it with the conclusions obtained from the Cooja simulator runs detailed in the previous section 4.9, and we will formulate a final thesis conclusion that will lead up to the “Future development” section.

### 5.1. Ciphers

Symmetric Key Cryptography ensures both confidentiality and authenticity.

Public Key Cryptography generates three scenarios:

- **NO** confidentiality, but authenticity
- Confidentiality, but **NO** authenticity
- Both confidentiality and authenticity

Going back to the questions raised in chapter 3.1 “*Is it really needed in an IoT network to perform an authentication validation by 3<sup>rd</sup> parties? Can we avoid this fact in order to make our devices faster, computationally speaking, by only using Symmetric Key Cryptography?*” as already stated in that section, cryptography (and obviously also in IoT) is not directly applied in the form of *Message-encryption-decryption* in the lonely form of a public or a private Key, it is composed of more complex structures. But focusing on **the 3<sup>rd</sup> party authentication** context the reality here is that **it will depend on the specific application**. IoT could be to almost every device able to track some data, so as we will further see, the importance of this data’s validation will mark if the information needs to be authenticated or not. Probably if I’m scheduling my oven to start cooking 1h before I get home there is no need to authenticate that I am actually sending this message. On the other hand if I want to maintain a smoke monitoring in some hotel rooms I would like to ensure that the data received from the smoke sensors has not been manipulated by a *Man in the middle attack*.

Focusing now on the cipher structure, as it was also commented in section 3.2, due to current devices power and capacity, the vast majority of Symmetric Key algorithms are based on **Block Ciphers**. That fact was the main reason to start studying them. As it can be seen in the various presented figures in chapter 3, the results of the simulations show that block cipher algorithm can perfectly fit with IoT devices. **All that we shall take into account is the specific hardware device limitations**. As it was shown in the study carried by Thomas Eisenbarth, the CLEFIA algorithm perfectly suits Lightweight Cryptography limitations. Indeed CLEFIA presents the best

Throughput vs Gate Equivalent ratio amongst all the ciphers presented in “*Lightweight Cryptography for the Internet of Things*” by Masanobu Katagi and Shiho Moriai. So basically, if the **Block Ciphers** performance results are good enough to be held in IoT devices “*Why should we consider Stream ciphers? Also, security is known to be stronger for Block Cipher cases why would we possibly want to take a step-back?*” This question, raised during chapter 3.2, illustrates the thoughts of many researchers as Stream Ciphers had received little attention, but the answer is that Stream Ciphers can be better suited for the most simple IoT devices, why should they be taken apart?

As seen in that same chapter, the proposed MICKEY stream cipher implementation while requiring more hardware resources than other AES algorithm stream ciphers, had a better than those mentioned AES implementations. Still those needed resources are **smaller than the Hardware resources needed for a Block Cipher** implementation, while the level of security is the necessary. That basically shows us how Stream Ciphers also fit with IoT devices. Just like the block Cipher case, **all that we shall take into account is the specific hardware device limitations**. That leads us to the following trivial but not less important conclusion:

In a general perspective, as far as the IoT device hardware limitations permit it, we would choose a Block Cipher encryption algorithm as the first option. As soon as the hardware limitations prevent us to implement a block cipher schema, we shall look for the most suitable algorithm from the stream cipher family. The threshold here is the throughput limitation. We could find ourselves in a situation in which a block cipher algorithm exceeds the HW limitations but the most hardware suitable Stream cipher algorithm doesn't fit the application throughput requirements. If no algorithm satisfying both cases were found, the only way to solve this issue would be downgrading the application requirements.

So, overall, the conclusion of this section would be the following: The expectations from an efficient cryptographic algorithm will differ depending on the specific application. It is very difficult to expect that a single implementation will satisfy all requirements. It seems clear that, as stated in the previous paragraph, the most important aspect for a hardware/software-efficient cryptographic algorithm is flexibility. Meaning that this algorithm could implement different type of specific architectures making it flexible enough to adapt to the exposed HW limitations or throughput requirements.

Finally, one fact shall be remembered. The comparison of designs is performed without taking into account Message Authentication Code (MAC) support (which is something that we recalled to be out of the cope of our thesis). Secondly, performance in terms of throughput only, will not address latency issues, something that may be critical in some IoT applications **this point is the most interesting thing to be considered in the future development chapter**.

## 5.2. SHA-3 algorithm and Hash study

As seen in section 3.3, the hashing technique solved a security scratch on **Public Key Cryptography** that would make an attacker able to supplant the sender's identity. By using the Hash function a digital sign is applied to the sent message. *Is this applicable to the IoT world? If the answer is yes, is it viable with scenarios of potential hundreds of devices sending digitally signed information to each other?* This question is directly assessed like the one regarding the 3<sup>rd</sup> party validation. Once again **it will depend on the specific application**. This may seem redundant, but reinforces the idea that the **IoT world is wide both in the device and the algorithm application contexts**. If no specific standard is to be set for all manufacturers (which hasn't been the case and doesn't seem to be in the next future) it will be the specific applications needs (plus the HW limitation) that will tell us whether if it makes sense to apply hashing (in the different algorithms manners) in an IoT network or not. All we should take into account when hashing is the "cost" parameter. As this parameter grows, the amount of work (typically CPU time or memory) necessary to compute the hash increases exponentially; this is one parameter to add up to the performance ecosystem of an IoT device. As long as the device can handle it, there is no reason to think that it shall not be applied if our application demands it.

As seen in the corresponding chapter 3.3, the study "*Efficient and Concurrent Reliable Realization of the Secure Cryptographic SHA-3 Algorithm*" by Siavash Bayat-Sarmadi presented some actual SHA-3 implementations. In this study the RERO-based approach, an algorithm suitable for resource-constrained applications like IoT devices, is presented and compared. The key idea behind the presented results is that various implementations of the SHA-3 algorithm can be developed and compared to the original implementation. It is proven that KECCAK is simple enough to be studied and presented in different HW implementations. At the same time the throughput levels obtained in these implementations suggest that this algorithm would be really suitable for those applications that need higher levels of speed (maybe addressing latency issues). Overall, though, **Symmetric Key** algorithms are lighter and, therefore, better suited for IoT networks.

Regarding the hash simulations, we did not take into account the "cost" parameter as all the function runs were performed with a computer (2,4 GHz Intel Core i5 processor with 4 GB 1067 MHz DDR3 RAM memory module), which obviously have more performance capabilities. It was successfully tested how the Hash message is created (using various types of input data). This hash is then encrypted using the sender's private key, creating the data signature. The message (data) would then be sent along with its own signature. So at the receiver's end all it has to do is Hash the received data with the same Hashing function and decrypt. We basically used a simple encryption test method to do the reversal process and validate how the hash works properly, as the hashed decrypted message equals the digital signature sent (hash).

### 5.3. Elliptic Curve Cryptography

To conclude with the conclusions chapter, we will review the results obtained from the Elliptic Curve Cryptography study.

As commented in section 3.4, there are actual implementations of ECC algorithms in specific sensor platforms (for example MICAz Mote), directly implying that ECC implementations are feasible. As we have seen in the various figures it is a proven fact that ECC has improved its performance through the years by the implementation of different techniques. The implemented optimizations showed in that section, allow performance gains but also generate a collateral effect on memory consumption. As it was displayed in the correspondent figure, memory requirements for both the code size and RAM memory for the new more efficient implementations are increased. It is a logical conclusion to remark that a threshold defining what shall be considered as a limitation in each design parameter must be defined. This is also something to be taken under consideration in the **in the future development chapter**.

Regarding the executions of cryptographic protocols for key agreement and digital signatures tests, another question was raised *How do we know if these performance results are good enough for the IoT world? How can we compare it to the Ciphers performances?* The results showed in figures 37 and 38 referring to Dr. Malan's work shows us that in terms of latency the ECC results are equal to the Block Ciphers algorithms resented in previous sections. As soon as we start improving the device's properties this latency may improve, but it will come along with more HW requirements to fulfil these needs. Finally, as it was expected, the throughput levels of this Cryptographic technique are substantially better than the ones seen in the Ciphers sections, making this applications more than suitable for those applications that need higher levels of speed, just as said in the previous section 4.3. It is important to recall once again that overall, **Symmetric Key** algorithms are lighter and, therefore, better suited for IoT networks, specially for those devices needing the lightest possible computational charge. Those would be the example of sensors expected to have a usable life of various years, etc.

### 5.4. Overall conclusions

As it has been observed during the whole project, current cryptographic algorithms are ready and suitable to use in IoT devices in **terms of performance**. Each application and device will have its own limitations, so it is complicated to find a specific algorithm that will ensure correct performance while fulfilling all the limitations. That is the main reason why a closed solution has not been yet set. It is very difficult to expect that a single implementation will satisfy all requirements; the ideal algorithm shall be able to engage a wide range of architectures. Also, the three main performance characteristics seen in this project through the different chapters (computational cost, data transmission and battery cost) are relatively easy to quantify, while the remaining attributes (Flexibility/Scalability/Pipelining and Simplicity/Completeness/Clarity) that were not treated in this project, are much more subjective and will, once again, depend on every application and device.

Actually, this **smoothly corresponds with the results obtained from the Cooja simulator**. Under the same encryption standard, through various runs of simulations we did not see a major difference when applying encryption in both the Z1 and Sky motes. We could definitely see how encryption basically implied more power consumption for the motes but almost the same data

transmission rate. For example, let's remember that our study framework dealt with no data authentication; That may be a critical point for some IoT applications, while others can absolutely pass on this feature. This generates the obvious idea that each application shall be tested according to their specific needs and that is the best way to discern between the data ciphering necessity, power consumption restriction and information transmission latency. In other words, if I have developed an IoT application that collects temperature data from each room in a house and uses all this information to generate a real time condition map in order to adjust the heating, it may seem obvious that neither information transmission rate nor encryption are important features for this application. It is not critical if the data from each sensor is transmitted to the heat controller in 100ms or 10s, as the house temperature never changes this fast, so that will not affect the application performance. Also, there is no need to encrypt the data as no valuable information could be obtained from a room's temperature. On the other hand, an application optimized in the field of power consumption via software and hardware performance is key, as we would not be willing to change the sensors battery (or directly the device itself) each month.

This specific example could be turned all the way around. We may be willing to gather confidential data from a closed framework and store it, for example, in a central cloud infrastructure. Under this premise, data encryption would come to the top and both software and hardware shall be optimized for data protection while probably losing battery optimization.

As a personal point of view, it seems that nowadays the majority of people does not yet conceive the IoT concept. It still has not been implemented in our daily life, so that is why especially in the case of small devices the solutions are still wide open, something that may not help to integrate different devices from different companies with different protocols in the same IoT network. Those devices that do not present either size restrictions or computational limitations (Autonomous cars, household appliances, public traffic control devices, etc.) will actually work as if they were computers and will easily transition into the IoT world. But as it was being said, those small devices (which were the central point of our investigation) will be the ones presenting more problems when it becomes all integrated.

Finally, in a learning standpoint, I have to say that it was interesting to get introduced to the cryptography world while exploiting and investigating its application in the IoT. As it was said in the objectives section, cryptography is a topic that was not exploited neither in the degree nor in the master so it was nice to take advantage of the master thesis to explore it and get introduced to it.





## 6. Future development

After performing this thesis the three following points were thought to be interesting to develop (following this line of work):

1. The comparison of designs was performed without taking into account Message Authentication Code (MAC) support (which is something that we recalled to be out of the scope of our thesis). Also, performance is expressed in terms of throughput only. That does not address latency issues, something that may be critical in some IoT applications. The study of the latency affectation related with the throughput level for some application should be really interesting as it is something that would badly downgrade the user's experience.
2. A threshold defining what shall be considered as a limitation in each design parameter must be defined. That is something that I have not seen done. A full research (or group of researches) exposing each possible limitation design points and defining what is an acceptable threshold for each one would also be interesting, specially from an academic point of view. That would help to define a close framework of what an IoT application device should expect under a closed set of conditions. With that being said, it is also important to recall how difficult this is, as that ultimately depends on each application expected features.
3. Finally, once the documentation part of the project has been carried. A project could be started directly on the emulation chapter. Following the same idea of an enclosed framework, it could study variations related to data validation, data encryption using different key sizes in the same exact scenario, data throughput limitation, etc. That would be like a simulation for a specific application needs. The enclosed framework could be something like: *"We want an application that ensures X level of data protection (that could be protection against a concrete attac) while providing a Y life cycle: Which is the best combination (HW + ciphering schema) for this application?"*



## 7. Bibliography

- i. Lee, J. Y., Lin, W. C., & Huang, Y. H. (2014, May). A lightweight authentication protocol for internet of things. In *Next-Generation Electronics (ISNE), 2014 International Symposium on* (pp. 1-2). IEEE.
- ii. A Eisenbarth, T., & Kumar, S. (2007). A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6).  
  
He, D., & Zeadally, S. (2015). An analysis of rfid authentication schemes for internet of things in healthcare environment using elliptic curve cryptography. *IEEE internet of things journal*, 2(1), 72-83. .
- iii. Hatzivasilis, G., Theodoridis, A., Gasparis, E., & Manifavas, C. (2014). An Ultra-lightweight Cryptographic Library for Embedded Systems .
- iv. Kumar, U., Borgohain, T., & Sanyal, S. (2015). Comparative Analysis of Cryptography Library in IoT.
- v. Malan, D. (2004). *Crypto for tiny objects*. Technical Reprot TR-04-04, Harvard University.
- vi. Faquih, A., Kadam, P., & Saquib, Z. (2015, September). Cryptographic techniques for wireless sensor networks: A survey. In *Bombay Section Symposium (IBSS), 2015 IEEE* (pp. 1-6). IEEE.
- vii. Bayat-Sarmadi, S., Mozaffari-Kermani, M., & Reyhani-Masoleh, A. (2014). Efficient and concurrent reliable realization of the secure cryptographic SHA-3 algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(7), 1105-1109.
- viii. Shokrollahi, J. *Rheinischen Friedrich-Wilhelms-Universitat Bonn* (2006).
- ix. Aranha, D. F., Dahab, R., López, J., & Oliveira, L. B. (2010). Efficient implementation of elliptic curve cryptography in wireless sensors. *Adv. in Math. of Comm.*, 4(2), 169-187.
- x. Liu, Liu, Z., Großschädl, J., Hu, Z., Järvinen, K., Wang, H., & Verbauwhede, I. (2017). Elliptic curve cryptography with efficiently computable endomorphisms and its hardware implementations for the internet of things. *IEEE Transactions on Computers*, 66(5), 773-785.
- xi. Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, 48(177), 203-209..
- xii. Kermani, M. M., Zhang, M., Raghunathan, A., & Jha, N. K. (2013, January). Emerging frontiers in embedded security. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on* (pp. 203-208). IEEE.
- xiii. Baldwin, B., Byrne, A., Lu, L., Hamilton, M., Hanley, N., O'Neill, M., & Marnane, W. P. (2010, August). FPGA implementations of the round two SHA-3 candidates. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on* (pp. 400-407). IEEE.
- xiv. Rogawski, M. (2007). Hardware evaluation of eSTREAM candidates.
- xv. Good, T., & Benaissa, M. (2007). Hardware results for selected stream cipher candidates. *State of the Art of Stream Ciphers*, 7, 191-204.
- xvi. Malan, D. J., Welsh, M., & Smith, M. D. (2008). Implementing public-key infrastructure for sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(4), 22.

- xvii. Zhang, Z. K., Cho, M. C. Y., Wang, C. W., Hsu, C. W., Chen, C. K., & Shieh, S. (2014, November). IoT security: ongoing challenges and research opportunities. In *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on* (pp. 230-234). IEEE. .
- xviii. Katagi, M., & Moriai, S. (2008). *Lightweight cryptography for the internet of things*. Sony Corporation, 7-10.
- xix. Kitsos, P. (2005). *On the Hardware Implementation of the MICKEY-128 Stream Cipher*. IACR Cryptology ePrint Archive, 2005, 301.
- xx. Babar, S., Stango, A., Prasad, N., Sen, J., & Prasad, R. (2011, February). Proposed embedded security framework for internet of things (iot). In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on* (pp. 1-5). IEEE.
- xxi. Rao, M., Newe, T., & Grout, I. (2014, September). Secure hash algorithm-3 (SHA-3) implementation on Xilinx FPGAs, suitable for IoT applications. In *8th International Conference on Sensing Technology (ICST 2014)*, Liverpool John Moores University, Liverpool, United Kingdom, 2nd-4th September.
- xxii. Roman, R., Najera, P., & Lopez, J. (2011). Securing the internet of things. *Computer*, 44(9), 51-58.
- xxiii. Zhang, T., Zheng, Y., Zheng, R., & Antunes, H. (2016). *Securing the Internet of Things- Need for a New Paradigm and Fog Computing*, Corporate Strategic Innovation Group, Cisco Systems, Inc., San Jose, CA, USA.
- xxiv. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., & Iwata, T. (2007, March). The 128-bit blockcipher CLEFIA. In *International Workshop on Fast Software Encryption* (pp. 181-195). Springer, Berlin, Heidelberg.
- xxv. Mahalle, P. N., Prasad, N. R., & Prasad, R. (2014, May). Threshold cryptography-based group authentication (TCGA) scheme for the internet of things (IoT). In *Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE), 2014 4th International Conference on* (pp. 1-5). IEEE.
- xxvi. Alcaraz, C., Najera, P., Lopez, J., & Roman, R. (2010). Wireless sensor networks and the internet of things: Do we need a complete integration?. In *1st International Workshop on the Security of the Internet of Things (SecIoT'10)*.
- xxvii. Team Keccak. (n.d.). Retrieved from <https://keccak.team/keccak.html> - Technical details table, KECCAK webpage.
- xxviii. Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2007, May). Sponge functions. In *ECRYPT hash workshop* (Vol. 2007, No. 9). – Chapter 2.2: The sponge construction.
- xxix. Biham, Eli and Shamir, Adi (1991). "Differential Cryptanalysis of DES-like Cryptosystems". *Journal of Cryptology*. 4 (1): 3–72. Retrieved from [https://en.wikipedia.org/wiki/Data\\_Encryption\\_Standard](https://en.wikipedia.org/wiki/Data_Encryption_Standard) - The Feistel (F) Function
- xxx. Sony Global - CLEFIA - About CLEFIA - Key Schedule / Components- DoubleSwap Function. Retrieved from <https://www.sony.net/Products/cryptography/clefia/about/element.html>.
- xxxi. Apostolos, P. (2009). *Cryptography and Security in Wireless Sensor Networks*. FRONTS 2nd Winterschool Braunschweig, Germany. – Chapter 1: Elliptic Curve Cryptography (1/2)
- xxxii. The Z1 mote · Zolertia/Resources Wiki – GitHub. Retrieved from <https://github.com/Zolertia/Resources/wiki/The-Z1-mote>
- xxxiii. Contiki: The Open Source Operating System for the Internet of Things. Retrieved from <http://www.contiki-os.org>
- xxxiv. IoT operating systems – Devopedia. Retrieved from <https://devopedia.org/iot-operating-systems>
- xxxv. Simulation - How to enable message encryption in Contiki / Cooja. Retrieved from <https://stackoverflow.com/questions/37382634/how-to-enable-message-encryption-in-contiki-cooja-simulator>
- xxxvi. Github `contiki-os/contiki/core NONCORESEC` repository. Retrieved from <https://github.com/contiki-os/contiki/tree/master/core/net/llsec/noncoresec>

- xxxvii. The Official Contiki OS Blog: A big step for Contiki: built-in encryption. Retrieved from <http://contiki-os.blogspot.com.es/2014/10/a-big-step-for-contiki-built-in.html>
- xxxviii. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., & Markov, Y. (2017, August). The first collision for full SHA-1. In Annual International Cryptology Conference (pp. 570-596). Springer, Cham.
- xxxix. Symmetric Key Cryptography in Use, Alice and Bob - Image. Retrieved from: <http://securitycerts.org/review/symmetric-key-in-use.htm>
- xl. Main Differences Between Symmetric and Public Key Cryptography – Image. Retrieved from <http://www.jayitsecurity.com/2013/01/main-differences-between-symmetric-and.html>
- xli. Fredman, Michael L.; Komlós, János (1984), "On the size of separating systems and families of perfect hash functions", *SIAM Journal on Algebraic and Discrete Methods*, 5 (1): 61–68. Retrieved from [https://en.wikipedia.org/wiki/Perfect\\_hash\\_function](https://en.wikipedia.org/wiki/Perfect_hash_function)
- xlii. Team Keccak. (n.d.). Retrieved from <https://keccak.team/keccak.html> - Technical details table, KECCACK website



## 8. Economical study

Taking figure 1 as a basis, adding the time deviations, the economical study is presented in the following image:

| CONCEPT  | DETAIL                           | NUMBER OF HOURS | COST/HOUR | FINAL COST     |
|--|----------------------------------|-----------------|-----------|----------------|
| Documentation<br>(redaction and review)  | Project Proposal                 | 10              | 10 €      | 100 €          |
|  | Critical Design Review           | 10              | 10 €      | 100 €          |
|  | Result analysis                  | 30              | 10 €      | 300 €          |
|  | Final Report                     | 40              | 10 €      | 400 €          |
|  | Thesis Presentation              | 15              | 10 €      | 150 €          |
| Investigation about the<br>Topic:<br>Cryptography<br>Stream/Block Ciphers<br>ECC / Cooja simulator | Documentation about Cryptography | 30              | 10 €      | 300 €          |
|  | State of the art                 | 25              | 10 €      | 250 €          |
|  | Stream Cipher studies            | 15              | 10 €      | 150 €          |
|  | Block Cipher studies             | 15              | 10 €      | 150 €          |
|  | ECC studies                      | 15              | 10 €      | 150 €          |
|  | Learnnging about software        | 10              | 10 €      | 100 €          |
| Programming of the<br>simulations  | First simulation tests           | 15              | 10 €      | 150 €          |
|  | WSN structure design             | 20              | 10 €      | 200 €          |
|  | Mote emulation documentation     | 10              | 10 €      | 100 €          |
|  | Non-enciphered Simulations       | 25              | 10 €      | 250 €          |
|  | Enciphered Simulations           | 25              | 10 €      | 250 €          |
| Software costs   | Software license                 | ---             | ---       | 0 €            |
|  | Software amortization (5%)       | ---             | ---       | 0 €            |
| Study of results   | Result analysis                  | 35              | 10 €      | 350 €          |
|  | Draw conclusions                 | 10              | 10 €      | 100 €          |
|  | Further Research                 | 10              | 10 €      | 100 €          |
| <b>TOTAL</b>   |                                  | <b>365</b>      |           | <b>3.650 €</b> |





## 9. Appendices

In this section the simulation runs code and output is attached.

### 9.1. Log listener files

The four log listener files will be attached as PDFs files independently from this Memory paper. The files are named the following:

1. Loglistener.pdf
2. loglistener2.pdf
3. loglistener3.pdf
4. loglistener4.pdf

### 9.2. CSC files

The CSC format is the one used by Cooja simulator. This format can be converted into a plain text file and then into a PDF. Following the same idea than with the Log listener files they will be attached as PDFs files independently from this Memory paper. The CSC files are named the following:

1. simulation-1-sink-sender.pdf
2. simulation-2-sink-sender.pdf
3. simulation-3-udp-rpl-broadcast-example.pdf
4. simulation-4-udp-rpl-broadcast-example-PowerTrace.pdf
5. simulation-5-udp-rpl-broadcast-z1-example-PowerTrace.pdf
6. simulation-6-udp-rpl-broadcast-sky-example-crypto.pdf
7. simulation-7-udp-rpl-broadcast-z1-example-crypto.pdf
8. simulation-8-sink-sender-sky-crypto.pdf
9. simulation-9-sink-sender-z1-crypto.pdf