Security in Community Sensor Networks

Master's Thesis

Defense date: February 2, 2015

Unai Perez Goya

Supervisor:
Manel Guerrero Zapata
Departament d'Arquitectura de Computadors

Master in Innovation and Research in Informatics
Computer Networks and Distributed Systems

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

In this Master thesis, I will introduce CommSensum, a community sensor network developed by CompNet research group. I will explain what are the needs that led to creating the platform, how is the platform until now and I will focus on the security issue. As it is a platform that is beginning its first deployment tests, security issues had been sidelined. But now, the first steps need to be done.

Therefore, this thesis will analyze the various security options that exist, I will analyze which of these options are able to be adapted to the sensor network environment. Also, I will explain how the main bottleneck to provide of security to this type of platform ithe nodes own hardware limitation. I will analyze two main factors in the system to understand why there is no possibility to use HTTPS, RN171 WiFi module and mote's limitations. In addition I will compare already existing work, how other researchers design different wireless sensor network structures to give a solution. Some tricks can be done having a proxy or a intermediate router. Once immersed in the topic, I will analyze the components of CommSensum and see what security schemes I can develop.

Finally, I will also provide the functions needed to implement secure connections between the nodes and the server. All of these function will run on the mote and with its hardware limitations. I will provide too, some function using RSA to use in future in a future Waspmote version or another Arduino board because in the actual version there is not enough SRAM.

Finally, I presents some conclusions and future work possible to continuing implementation of the security in CommSensum platform.

Contents

1 Introduction **1**
- 1.1 Introduction . . . 2
- 1.2 CommSensum . . . 2
- 1.3 Internet of Things . . . 3
- 1.4 Server . . . 3
- 1.5 Nodes . . . 4
- 1.6 Thesis objective: System security . . . 4
  - 1.6.1 Server Security . . . 5
  - 1.6.2 Nodes Security . . . 6

2 Related work **8**
- 2.1 Analyses Works . . . 9
  - 2.1.1 Security types . . . 9
  - 2.1.2 Embedded works . . . 12
- 2.2 What can be implemented . . . 14
- 2.3 Ideal Case . . . 15
  - 2.3.1 RN171 . . . 15
  - 2.3.2 Alternatives . . . 16

3 Analyze and Prepare Test Environment **17**
- 3.1 Existing technologies . . . 18
  - 3.1.1 NodeJS . . . 18
  - 3.1.2 Django . . . 18
  - 3.1.3 Nodes . . . 18
- 3.2 Creating test environment . . . 19
  - 3.2.1 Server deployment . . . 20
  - 3.2.2 Linksys WRT54G . . . 20
  - 3.2.3 Configuration RN171 . . . 21
  - 3.2.4 System control . . . 22
  - 3.2.5 Waspmote IDE . . . 24

4 Development and Results **25**
- 4.1 Introduction . . . 26
- 4.2 AES implementation . . . 26
  - 4.2.1 Libellium libraries . . . 26
  - 4.2.2 NodeJS libraries . . . 27
  - 4.2.3 Connection between NodeJS and Waspmote . . . 27

4.3 RSA implementation . . . 32
4.3.1 Waspmote development . . . 32
4.3.2 NodeJS development . . . 35
4.4 Results . . . 36
4.4.1 Waspmote . . . 36
4.4.2 Arduino Yun . . . 37
4.4.3 More capacity hardware . . . 37
4.5 Integration on CommSensum . . . 37
4.5.1 New Data Base tables . . . 38
4.5.2 Changes on Frontend . . . 38

5 Conclusions and Future work **39**
5.1 Conclusions . . . 40
5.2 Future work . . . 41

Bibliography **42**

List of Figures

1.1 Composition of CommSensum. 3
1.2 IoT application domain. 4
1.3 CommSensum platform structure. 5
1.4 Waspmote with WiFi module. 6

2.1 Block cipher modes comparison. 10
2.2 SIZZLE architecture. 12
2.3 SSNAIL architecture. 13
2.4 Libelium offered solution. 14
2.5 Nodes connect directly to API server without intermediaries. 14
2.6 Libelium offered solution Meshlium. 15

3.1 Linksys WRT54G integrated router. 21
3.2 RN171 WiFi module. 22
3.3 Test environment structure. 23

4.1 How standars proliferate. 28
4.2 Arduino yun architecture. 37
4.3 New data base tables. 38

List of Tables

2.1 Embedded nodes security comparison. 12

3.1 Waspmote hardware characteristics. 19
3.2 Different Arduino comparison table. 19
3.3 Test environment server characteristics. 20

Listings

2.1 Code to get modulus and exponents. 10
3.1 Code to configure WiFi module using Raspberry Pi shield. 21
3.2 Script to manage 2 connections in linux. 23
4.1 From uint8_t to hexadecimal char array function. 26
4.2 AES encryption function. 26
4.3 Encrypt using NodeJS libraries. 27
4.4 Encryption using Node-Forge. 29
4.5 Needed definition in waspmote. 29
4.6 Encrypt using AES in waspmote. 29
4.7 Send AES encrypted data in waspmote. 30
4.8 Encrypt using RSA function in waspmote. 31
4.9 Decript AES function on NodeJS. 31
4.10 Necesary defines on CSApiHelper. 32
4.11 Random password generate function. 32
4.12 Encrypt using RSA function in waspmote. 33
4.13 Waspmote digital signature function. 33
4.14 Encrypting data array using RSA in waspmote. 34
4.15 Send RSA data from waspmote. 35
4.16 decryptRSA function. 35

Chapter 1

Introduction

Contents

1.1 Introduction **2**

1.2 CommSensum **2**

1.3 Internet of Things **3**

1.4 Server **3**

1.5 Nodes **4**

1.6 Thesis objective: System security **4**

1.6.1 Server Security 5

1.6.2 Nodes Security 6

1.1 Introduction

If we take a look in different European or World's big cities we will see the increasing of pollution that are not good for people because of worsening health. Different pollution types (air, noise, light, etc...) are normal in these cities and the first step to combat this is to be aware. For this, it is necessary to measure the pollution.

At the same time that these harmful elements increases we can see how society faces this problem. Ecology sentiment is growing as we realize that we have the responsibility of this pollution. In different cities, politicians try to create a response for that need to measure this pollution. That is why different cities are deploying or already have deployed different systems to measure and publish this pollution. Some ecologist groups claims, that some governments try to minimize the pollution measures removing sensor nodes on zones that have high pollution level [6] [12] [4].

That is why, during the last years urban participatory sensing has gained strength, or put another way, the idea that individual persons or community groups can sense the environment. In order to, send the data to centralized servers and create applications using this data. Like it says in the CommSensum project web [26]*"The objective of CommSensum [27] is to build an Open Link Data (OLD) Community Sensing Platform . The Platform will have a real use in the context of Smart Cities"*. Based on this idea of being independent from different interest, The CommSensum project has been created. This is the response given by CompNet Research Group [5] of Facultat d'Informàtica de Barcelona (FIB) [9]

1.2 CommSensum

With transparency in mind, the biggest challenge of this project is to provide all citizens the ability to buy some hardware with the needed software and run its own node. Besides the node it also needs an Internet connection to send measurement data to the central server. Among all, we can create a big set of data from the city.

That does not mean that only can take individual or community groups data. On the Internet there are published a lot of public or open data from governments or different community groups that can be used too. In this way, we can get as much valid data as possible and many of them in real time. The project is already online [27] and it has a mobile app [1] financed with microdonations, example that there are people who want to finance these type of projects.

CommSensum has two main parts to work, server and nodes. The server side is the part that need to be implemented by one association or govern. The nodes side even if is deployed by association or govern, can be completed too by individual users or just be a user network.

Up to now, the development of this product has been done to work properly using different types of nodes associated to an API server. The API server was created taking into account web service architecture, that is, new versions can be developed maintaining previous ones.

This does not mean that is the only way to use this type of technologies. Libelium's web page, we can see other environments to use it like, Detecting Radiation Levels in

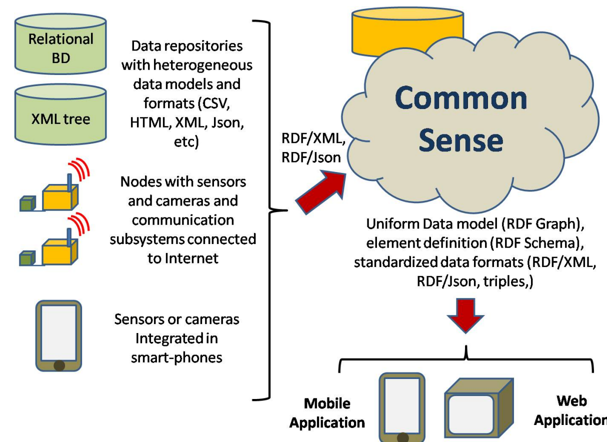


Figure 1.1: Composition of CommSensum.

Fukushima, smart Agriculture project in Galicia and other interesting application that based on meaning the environment [13].

1.3 Internet of Things

Recently has been started to introduce Internet of Things (IoT) term. A lot of paper and survey have been written to explain and define this. One example is given by Luigi Atzori in [30] he say: *The Internet of Things(IoT) is a novel paradigm that is rapidly gaining ground in the scenario of modern wireless telecommunications*. But he is not he only one that tries to define, Gerd Kortuem and Fahim Kawsar did a definition of IoT on [35] like this: *"The combination of the Internet and emerging technologies such as near- field communications, real-time localization, and embedded sensors lets us transform everyday objects into smart objects that can understand and react to their environment"*. So different people explain IoT term like an evolution of existing technologies or "smart object" and its combination with Internet. They talk about how the reduction in terms of size, weight, energy consumption, and cost of the radio is going to be essential on this technology evolution to create new big platforms and applications taking advance of this benefits.

On figure 1.2 we can see how see Luigi Atzori on [30] the different application for IoT and we can derive that a system like CommSensum is not considered but a lot of them have relation in the fact that they sense the environment.

1.4 Server

CommSensum server is composed by two independent parts, the backend and frontend [27]. Taking into account that, the system needs to be prepared to have a wide number of connection is the more logical structure.

The backend, was created using NodeJS [18]. This is the API of the platform, always will be running waiting for arrival data. It is the part of the system that distinguish different data and user to save on the database.

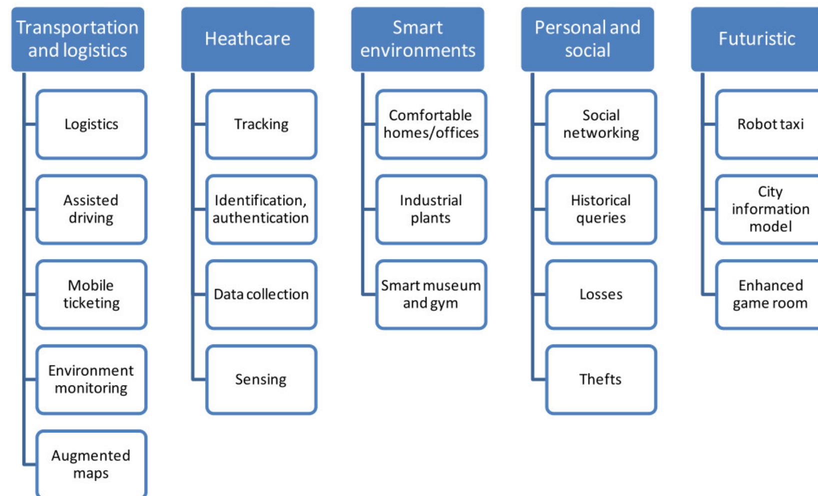


Figure 1.2: IoT application domain.

The frontend has been built with Django [8] framework and is used to create and add to the database users, projects, new nodes, data strings, etc... At the same time, this part can shows all data of different existing projects and all data of the project whenever user have permissions.

1.5 Nodes

Like mentioned before, the idea of this project is to give the ability for people to send data using any hardware or software they want (both free or proprietary). But at the same time, is need give a official solution that is tested that works properly.

This solution can be different for each situation, that is, each problem is going to have a different solution. For example, is not the same to have the node connected to the grid or have it run on batteries. Because if you need to use batteries the system need to control power consumption and you can have a hardware barrier.

1.6 Thesis objective: System security

Once all the system works, it is the time to look for the security of the system. To ensure that all platform is secure all information movement need to be considered, from node to final user taking a project information. In other words, the system needs to be secure in communication between the user and frontend, in data storage on the server and in communication between node and API server. On figure 1.3 we can see the offered structure on CommonSense.

The first one will be done using HTTPS/SSL protocol because all user will need a computer to connect to the frontend and nowadays all computer has HTTPS support

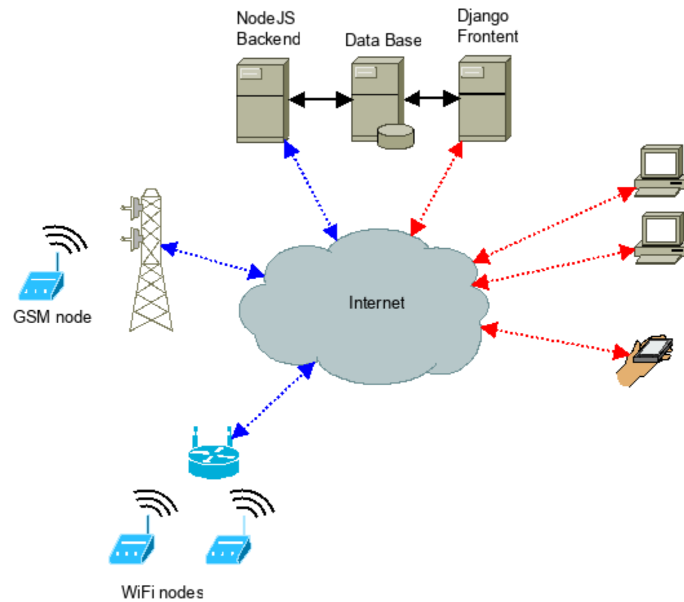


Figure 1.3: CommSensum platform structure.

because it is a widely extended method.

The security of the database has two pillars: server protection and definition of what permission have each user with projects and data protection that is very related with the main security of the server.

On communication between node and API server two aspects need to be considered: data protection and user authentication.

Data protection: The owner need to have the option to encrypt data that his sensor measures if he want to keep this information in private.

User authentication: All data server gets need to signed by the sender to ensure that it is a legitimate data sent by registered user.

In this thesis we will analyze in deep and give a solution to this third part, the security needed in the communication between the nodes and the server. The first step to implement this functionality is define what parts of CommSensum need to have some changes to implement this new functionality. If we want to analyze what changes are needed to implement, the best way is show the route of information from final user to nodes and backward.

1.6.1 Server Security

Our objective is to crypt all information sent by the node. But for that, we will pass through all system starting how user defines the security of his node; to continue we will define how all security modes are implemented on the server frontend such as, API server; we will finish explaining why different node types are used in this system and how we can implement different levels of security in each one. To understand better what we need to change, we will see parts of CommSensum, that needs some implementation for security improvement.

Frontend changes

When a user want to record one new node the form need to show server supported crypt types. Databases will need changes to have a record for that. Also new tables need to be designed to save passwords an more encryption information. At the same time new form type need to be design so the user has the responsibility to configure this encryption type.

API Changes

In the API part new services will be defined, this services when a data string arrives need to identify what user is sending information and if this information will be encrypted. For that, it needs to obtain the decrypt information from database, also if it is not encrypted, and decrypt data to save using the method already developed.

1.6.2 Nodes Security

The main testbed is going to be on nodes part, because we will have different types of nodes. For example, if you are going to place a node with electrical connection you can put more powerful hardware that if you want to place with batteries.

Have an electrical connection means that placed hardware can have more ram memory and better CPU to run applications. On this types of boards we can implement better security algorithms without hardware bottleneck. Security things we can implement on this type of boards will be significantly better.

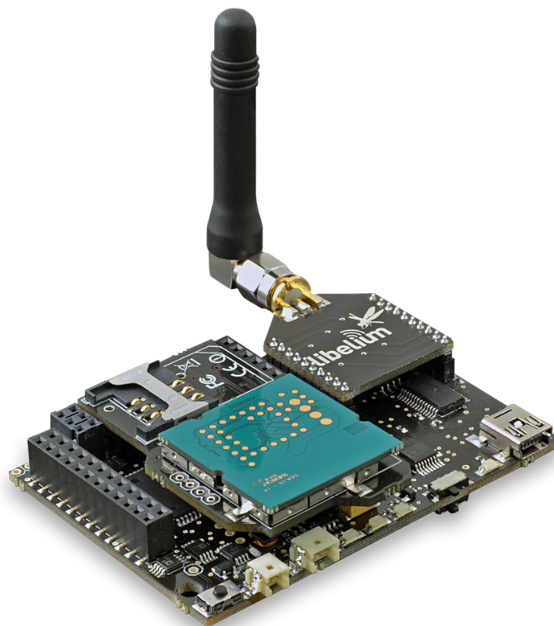


Figure 1.4: Waspmote with WiFi module.

But if electrical connection is not possible, it is necessary to use batteries. With this problem, in CommSensum uses Libelium hardware, already developed low power

consumption hardware. Libelium [14], sells different types of nodes, all defined to low consumption and to take different measurements. After different test to ensure that works properly, Waspmote [29] model was selected. We can see Waspmote board, on figure 1.4. Waspmote use ATmega1281 microcontroller with 14MHz frequency and 8KB of SRAM, clearly all we want to implement in this board needs to be carefully analyzed.

Chapter 2

Related work

Contents

2.1 Analyses Works **9**
2.1.1 Security types 9
2.1.2 Embedded works 12
2.2 What can be implemented **14**
2.3 Ideal Case **15**
2.3.1 RN171 15
2.3.2 Alternatives 16

2.1 Analyses Works

When we talk about network security, always we refer to same type of security used in the most web pages that have security needs. We talk about HTTPS. But we need to take into account that we will work with embed nodes and apart from normally used HTTPS, we need to analyze different works for it.

That is why we need to have a little introduction of HTTPS, RSA (an asynchronous public-key cryptosystem used by HTTPS), AES and some works done in IoT on security topic.

2.1.1 Security types

AES

AES is based on the principle of substitution-permutation network [37]. It combines both substitution and permutation, and is fast in both software and hardware. It is for this reason that it is widely used on embedded system. AES is a variant of Rijndael [31] but instead of only 128 bits of block size, has been extended to use a key size off 192, or 256 bits too.

AES operates on a 4x4 column-major order matrix of bytes, termed the state. Most AES calculations are done in a special finite field.

The key size on AES cipher specifies the number of repetitions of transformation rounds. That is different for each key size.

- 10 cycles for 128-bit keys.
- 12 cycles for 192-bit keys.
- 14 cycles for 256-bit keys.

To encrypt AES have a several processing steps, with different stages that each of one is different depending on the encryption key. To get the original message, AES uses the same steps but backward. AES is a block-cipher encryption that cuts into blocks the message and cypher its block individually.

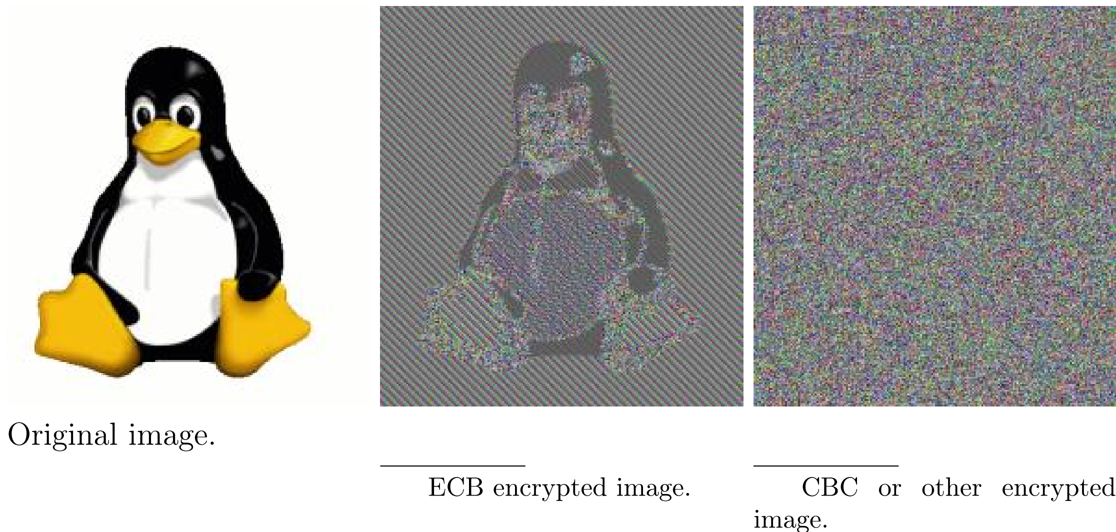
Exists different block cipher modes that can be used with AES: Electronic codebook (ECB), Cipher-block chaining (CBC), Propagating cipher-block chaining (PCBC), Cipher feedback (CFB), etc... But two main block cipher modes are used with AES, ECB and CBC of which CBC is the safer.

On figure 2.1 we can see and example of different block cipher modes applying it on a image. Even if, ECB photo has been encrypted, we still see the content of the image. However using CBC mode nothing can be seen.

RSA

RSA is the most extended asynchronous cryptographic system. This type of cryptography has two keys: public key and secret key. So you can encrypt data with one key that only can be decrypted with its complementary and vice versa.

The idea behind is that one of the keys will be published and the other will kept secret. To explain this we will use a typical example of Bob and Alice. Imaging that



Original image.

ECB encrypted image.

CBC or other encrypted image.

Figure 2.1: Block cipher modes comparison.

Alice publish a key and Bob wants to send an encrypted message to Alice. Bob will take Alice's key and it will encrypt the message with it. Only Alice will have the secret key to decrypt the message and see its content. But this only works if Bob only wants to send a encrypted message, what happens if at the same time Alice knows who is sending this message. Anyone can encrypt message with public key and impersonate Bob. To fix this issue the used method is signing the message.

To create RSA keys you need to follow different steps defines by Rivest, Shamir and Adleman [36]. There different elements are defined like, modulus, public and private exponents. Normally, when we talk about RSA we think in PEM format but since we have embed system, we will need modulus and exponents. On GNU/Linux, OpenSLL can be used to derive this elements using the next commands.

```
1  -------------------------------------------------
2  Create private and public key
3  -------------------------------------------------
4  openssl genrsa -out mykey.pem 1024
5  openssl rsa -in mykey.pem -pubout > mykey.pub
6
7  -------------------------------------------------
8  Show public modulus and exponent
9  -------------------------------------------------
10 openssl rsa -pubin -in mykey.pub -text -noout
11
12 -------------------------------------------------
13 Show private modulus
14 -------------------------------------------------
15 openssl rsa -noout -modulus -in mykey.pem
16
17 -------------------------------------------------
18 Show private exponent
```

```
19 --------------------------------------------------
20 openssl rsa -in mykey.pem -noout -text
```

Listing 2.1: Code to get modulus and exponents.

Signing message

To signing message instead of only one pair of key they will have two. We will see with Bob and Alice example to understand it. So now, Bob it will have a another pair of RSA keys and it will publish its public key. So when Bob want to send a message to Alice, first, he will diggest the message with message-digest algorithm like MD5 and will encrypt with his private key this MD5. This encrypted MD5 will be the signature of the message. After that he will encrypt both te MD5 and the signature, and send to Alice. So when Alice get the encrypted message and decrypt, it will get both packets. Now Alice can compare message MD5 diggesting at the receive moment and compare to bob signature MD5. If there are equal the message has been sent by Bob. The signature can be decrypted by any people but only encrypted by Bob. That means that only Bob can send this message.

HTTPS

Most people associate HTTPS with SSL (Secure Sockets Layer) which was created by Netscape in the mid 90's. But this is not totally true, because when Netscape disappear SSL's maintenance was moved to the Internet Engineering Task Force (IETF). With this move, the first version was re-branded as Transport Layer Security (TLS) 1.0. Let's see how it works.

Essentially HTTPS uses asynchronous cryptography like RSA to create a secure communication between client and server. This start with hello message sent from client.

Client Hello TLS wraps all traffic in “records” of different types. The first wrap sent by the client is a “handshake” record. This message means that client want to create a secure connection with the server.

This part of the communication is public every one can take this message. Among other things, this record sends the type of cryptography supported by the client.

Server Hello Server answer to this “handshake” with a record that agreed the first request. This record have three different sub-messages.

1. **Server hello**: In this part of the message server select what type of encryption they are going to used. This type will be selected from client sent list.
2. **Certificate Message**: Here server send the certificate so that the client can verify it.
3. **Server hello done**: This part, notifies that the message is a completed.

Checking out the Certificate The browser take certificate message and ensures that this certificate is in force today. It also checks to make sure that the certificate's public key is authorized for exchanging secret keys.

Although there are more steps, the main idea of HTTPS is that client and certificated server they use RSA to exchange session keys and cipher all exchanged data using a simpler encryption method, something like AES, using this random session key.

2.1.2 Embedded works

When you look for research on this topic you realize that several number of work has been done. OpenSSL [19], MarixSSL [20], GoAhead WebServer (Secure Embedded web server), Sizzle [32], SNAIL [33], and a large number of works more can be found [34]. Each of different solutions have different architecture or different hardware requirements. In order to compare these implementations on Table 2.1 we can see the memory needed for each of the schemas, the reason behind this is that SRAM is the bottleneck of many nodes like the Waspmote.

Name	Memory need [KB]
OpenSSL	1000
MarixSSL	50-70
GoAhead WebServer:	60
Sizzle:	less than 4
SSNAIL:	7

Table 2.1: Embedded nodes security comparison.

Nodes used in CommSemsun have 8KB on SRAM memory so, OpenSSL, MariSSL and GoAhead WebServer can be discarded. Sizzle and SSNAIL can be run on Waspmote nodes but, we can find two main differences, system architecture and SRAM memory need.

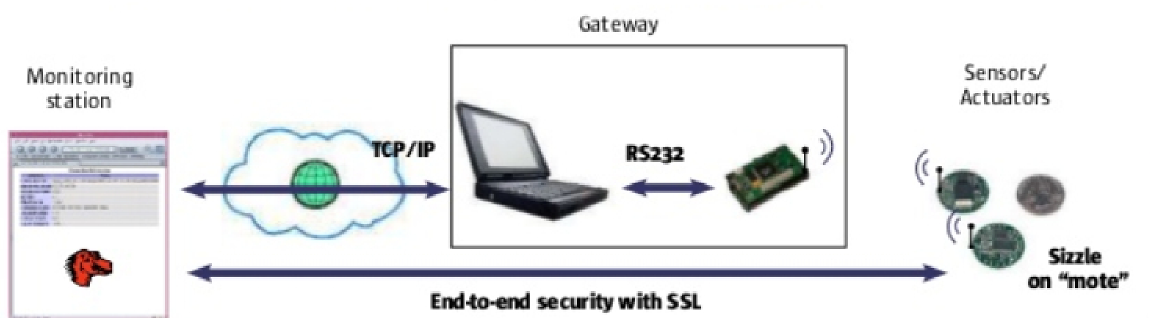


Figure 2.2: SIZZLE architecture.

The differences between our work and Sizzle is that has proxy based architecture like we see on figure 2.2. TCP/IP connection terminates at the gateway; and sensor network

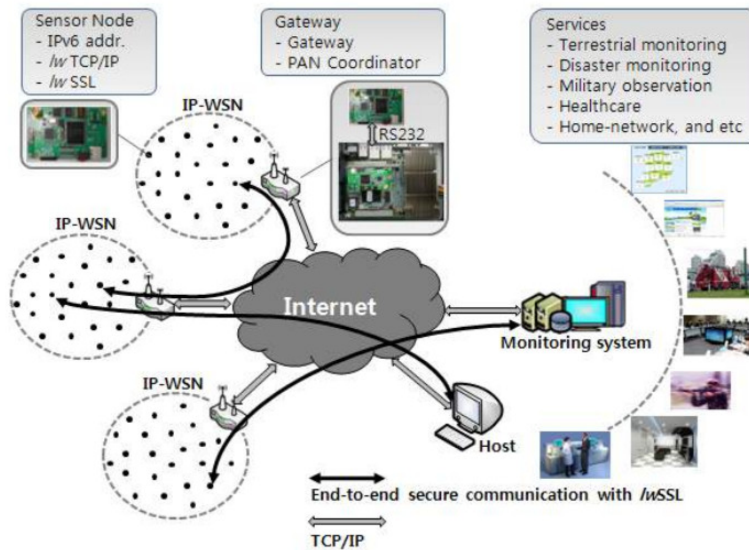


Figure 2.3: SSNAIL architecture.

using Sizzle uses its own protocol instead of IP-WSN standard. So, this proxy based architecture go against what we have.

In the figure 2.3 we see SSNAIL architecture, this is the same architecture offered by Libelium that we will see later. The problem to implement this solution in our system is that CommSensum have no IP-WSN gateway like in that example. Also we need to take into account that SSNAIL uses 7KB of SRAM, that is, the rest falls a bit short to run CommSensum code.

Although progress has been made on the issue of nodes security using SSL, today there is no possible to create a SSL connection on conditions we want. So, we need to measure our possibilities to implement system security.

2.2 What can be implemented

The main differences between work we analyze and CommSensum is that the main effort in this type of work is to create a sensor network were all sensors are connected between them only some of them have a connection to one router that is the unique element in the system that have access to the internet network. This router manages data and creates connection to the server without hardware bottleneck.

All nodes exchanges information to the edge of the system and edge nodes are the responsible to send information to the server using router. On figure 2.4 we can see Libelium offered router model to do that, Meshlium.

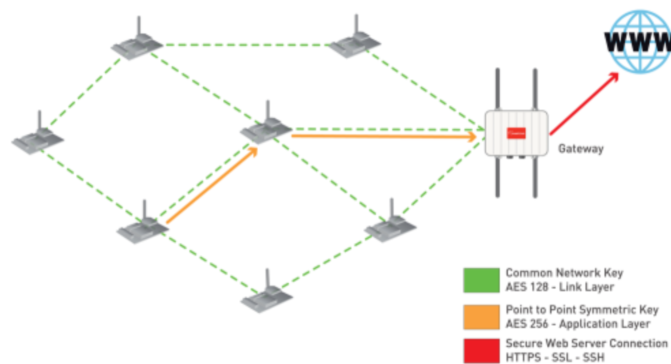


Figure 2.4: Libelium offered solution.

In this type of system the security between nodes is different from the security uses between router and server. On the first one, nodes uses AES, DES or other security type that no need high hardware resources. On the second one how is no need to take into account hardware limitations and they use HTTPS.

Our system is more like [34] work that uses SSL connections between nodes and server directly without using a specific router that it will send with HTTPS.

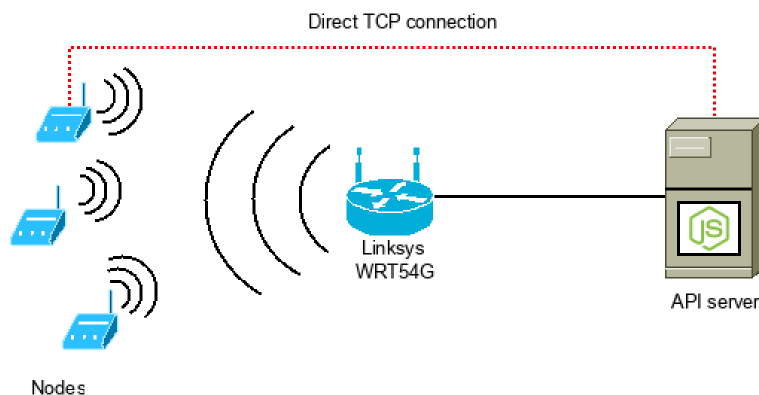


Figure 2.5: Nodes connect directly to API server without intermediaries.

On figure 2.5 we can see how both, CommSensum and this paper implementation, are the same problem because both structure are equal. We can expect that work done on [34] paper is the solution for our problem because they implement a SSL connection on embedded node system with 8KB RAM memory. In our system this type of nodes are used but we will see why this work is not enough for our case.

2.3 Ideal Case

The ideal case is to use HTTPS for all connection between nodes and server but we have a hardware bottleneck to do that. We see some case that they implement SSL on system with 4KB RAM but the memory used by the base application on CommSemsun only leaves us free 1KB ram before put already developed mentioned code, and that is not enough to run this implementation of SSL. Clearly is an important work but not valid to our case.

To connect Waspmote using wireless technology Waspmote uses WiFi module [15]. This module provides a Low-Power WiFi chip and completes the current connectivity possibilities enabling the direct communication of the sensor nodes with any WiFi router. This module chip is the RN171 [23], that is a small form factor, ultra-low power embedded TCP/IP module. Although it is a ultra low power, it have more processing power than Waspmote. That is, the initial idea was try to create HTTPS connection using this module, so we can implement security without sacrificing Waspmote hardware limitations. At the same time like this WiFi module only works when some data need to be sent the energy consumption would be very low too.

2.3.1 RN171

If we go to Libelium Waspmote technical guide put that HTTPS and FTPS protocols are supported by WiFi module but when you go to the WiFi networking guide [15] you can see how they create HTTPS connections. All nodes using WiFi modules connect to meshlium router and this router is the element that creates HTTPS connection.

To ensure that this module really it have not the ability to create HTTPS connections it is necessary to go to the root. If we go to Microchip web page, and look for RN171, we can try find if exist any command for HTTPS connection but there are not any.

Figure 2.6: Libelium offered solution Meshlium.

From here, the only step we can try with RN171 is to see if this module have the needed hardware to run HTTPS even if is not developed. To know definitely if WiFi module can create HTTPS connection we need to look its datasheet [25]. There is not defined that HTTPS will run, therefore, we expect that is not possible to run HTTPS on this module.

2.3.2 Alternatives

Hardware used now at the platform have not capacity to run HTTPS protocol but we can see that RN171 has nearly the specifications needed to run it. It is quite possible that in the next year can get new hardware with this capability. In this new scene the unique change will be change WiFi module and program to create an HTTPS connection directly to the server and send data with current code.

But for now we need to create a secure connection for the hardware that actually support this platform. So we need to analyze what are the alternatives provides by Libelium for secure connections on Waspmote.

HTTPS

Libelium only offers HTTPS connection using Meshlium router, that's means that there is no possibility to create an HTTPS connection directly between node and CommSensum server. The unique alternative for use HTTPS is mentioned before of new supposed WiFi module that support this secure connection.

AES

Waspmote have support to use AES on WaspAES libraries. This libraries are open source libraries that you can find on GitHub [28]. This library, in order to encrypt and decrypt, use another libraries called AVR-crypto-lib [3]. The libraries that are implemented support AES in ECB and CBC modes, the unique different between both are the content of the vector with the password is salted.

- ECB: IV salt vector is a vector of zeros.
- CBC: IV salt vector can be any integer array you want.

The CBC mode is clearly much safer than ECB mode that its use is not recommended. At the same time this libraries supports all AES key size mentioned before 128-bit, 192-bit and 256-bit.

RSA

For RSA another libraries are provider from Libelium, WaspRSA libraries. Compared with WaspAES another external libraries are used, because they use some tools taken from polarSSL [22] libraries to do RSA formulas.

Now we have an idea of what different tools and existing methods we can use. We are going to start with implementation part and different problems which have emerged.

Chapter 3

Analyze and Prepare Test Environment

Contents

3.1 Existing technologies **18**
3.1.1 NodeJS 18
3.1.2 Django 18
3.1.3 Nodes 18
3.2 Creating test environment **19**
3.2.1 Server deployment 20
3.2.2 Linksys WRT54G 20
3.2.3 Configuration RN171 21
3.2.4 System control 22
3.2.5 Waspmote IDE 24

3.1 Existing technologies

As I mentioned before the system is composed by different technologies distributed on server and nodes. On server side, two different platforms are used, NodeJS and Django. Now we will have a little introduction of these technologies. All different nodes used until now are programed on C++ but depending on the board we are programming for we will use different libraries.

3.1.1 NodeJS

Node.js [18] is an open source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript [11]. Is an asynchronous event driven framework used to build scalable network applications. The main advantage of NodeJS is the form of programming prepared to avoid dead-locking and the large number of modules it has. All these reasons are the base to create service oriented API.

3.1.2 Django

Django [8] is a open source web application framework, written in Python, which follows the model–view–controller architectural pattern [7]. Like is said in its web, Django, was designed to facilitate Web development "without needing to reinvent the wheel". Its main pillars are: Django was designed to be as quickly as possible, helps developers avoid many common security mistakes and the ability to quickly and flexibly scale.

3.1.3 Nodes

On CommSensum to lower power consumption Waspmote is used but at the same time is being tested with some Arduino models or Raspberry depending the place or needs. All nodes are embedded system and can runs C coding, therefore the security needs to be program on C. Even if, using Raspberry or Arduino in both RN171 WiFi module is going to be used.

Waspmote

The main advantage of Waspmote is the power consumption which has been greatly reduced. All Libellium products have been thought to run with batteries but that does not mean it is only for that. On table 3.1 we can see Waspmote hardware information and on image 1.4 how it is the board.

Microcontroller:	ATmega1281
Frequency:	14MHz
SRAM:	8KB
EEPROM:	4KB
FLASH:	128KB
SD Card:	2GB
Weight:	20gr
Dimensions:	73.5 x 51 x 13 mm
Temperature range:	$[-10C, +65C]$
Clock:	RTC (32KHz)

State	Consumption
ON:	15mA
Sleep:	55uA
Deep Sleep:	55uA
Hibernate:	0.7uA

Table 3.1: Waspmote hardware characteristics.

Arduino

Arduino [2] defines its own project as *"It's an open-source physical computing platform based on a simple micro-controller board, and a development environment for writing software for the board"*. Here we can find a wide number of different Arduino boards for different user and needs besides different components. Some of them have been selected for use on CommSensum: Uno, Mega 2560 and Yun.

For years Arduino has been evolving and creating new products with better hardware or improving its consumption. This is the reason by which it was selected to test on CommSensum. On table 3.2 can see the differences between models. Some are very different between them from CPU Speed to SRAM. That is the reason for was taken different models to different environments.

Name	Uno	Due	Mega 2560	Yun
Processor	ATmega328	AT91SAM3X8E	ATmega2560	ATmega32u4
Operating Voltage Input Voltage	5 V/7-12 V	3.3 V/7-12 V	5 V/7-12 V	5 V
CPU Speed	16MHz	84 MHz	16MHz	16MHz
Analog In/Out	6/0	12/2	16/0	12/0
Digital IO/PWM	14/6	54/12	54/15	20/7
EEPROM [KB]	1	-	4	1
SRAM [KB]	2	96	8	2.5
Flash [KB]	32	512	256	32
USB	Regular	2 Micro	Regular	Micro
UART	1	4	4	1

Table 3.2: Different Arduino comparison table.

3.2 Creating test environment

To start to develop and test different security libraries, a copy of CommSensum need to be deployed. In this test environment, we can add different security libraries to test and

add some changes, at the same time that we send wrong data from test node to check if both sides works, nodes and server.

3.2.1 Server deployment

To create a develop test environment there is no need to have powerful server, in this case, selected machine was a Pentium 4 with Ubuntu server. We can see on table 3.3 its main components.

Processor	Pemtium 4
Hard drive	80GB HDD
Ram Memory	2GB
OS	OS ubuntu server 14.04 LTS
Network adapter	Ethernet 10/100

Table 3.3: Test environment server characteristics.

One we have a Ubuntu Server clean machine several steps need to be done to run CommSensum. CommSensum has all this steps defined on a developer guide.

1. It is necessary to install all dependencies, for example to run Django the server needs to have python. All needed dependencies are: git, mysql-server, python-mysqldb, libmysqlclient-dev, python-pip and python-dev.

2. With all dependencies installed it is necessary to download the source code from repositories using git:*git clone urlhttps://www.ac.upc.edu/projects/commonsense/git/commonsens commsensum* (Project account needed for this step).

3. On source code, we have some DataBase backup an synchronize script so it can create project user and a database for it. Then restore the Data Base and synchronize.

4. When database is running the next step is install Django and all requirements. All requirements are on a text file that can be used to help install them with this command:*pip install -r requirements.txt*. To run frontent only need is to change CommSensum Data Base connection file. Executing *manage.py runserver* the front end will start running.

5. To have server side ready only just need to run API server based on NodeJS. So NodeJS and some external libraries need to be installed. To execute API only need to execute the following command *node server.js*

When test server was ready the part to define was gonna be the test network. The element selected to connect different hardware between them was Linksys WRT54G.

3.2.2 Linksys WRT54G

Linksys WRT54G is a integrated router that has the capacity to share Internet connections among several computers via 100 Mbit/s 802.3 Ethernet and 802.11b/g wireless data links.

Figure 3.1: Linksys WRT54G integrated router.

It also has 4 Ethernet connectors so you can connect different elements either by WiFi or cable.

It use DD-WRT OS to run, that is Open Source embedded operating system based on the Linux kernel. This router has all commands and configuration options we need to create our test network. On image 3.1 we can see this router image.

3.2.3 Configuration RN171

Before start testing security libraries it is necessary to configure WiFi module to connect Linksys router. The unique mode to configure this WiFi module is connecting in serial mode. After review RN171 programming guide [24] we develop this configuration code.

```
1  void loop()
2  {
3  while (Serial.available()>0) {}
4  // Enters in command mode
5  Serial.print("$$$"); check();
6
7  // Sets DHCP and TCP protocol
8  Serial.print("set ip dhcp 1\r"); check();
9  Serial.print("set ip protocol 2\r"); check();
10
11 // Configures the way to join the network AP, sets the encryption of the
12 // network and joins it
13 Serial.print("set wlan join 0\r"); check();
14 Serial.print("set wlan phrase PASSWORD\r"); check();
15 Serial.print("join WIFI_AP_NAME\r"); check();
16
17 // Sets the DNS address
18 Serial.print("set i h 0\r"); check();
```

```
19 Serial.print("set d n pruebas.libelium.com\r"); check();
20
21 //Configures HTTP connection
22 Serial.print("set i r 80\r"); check();
23 Serial.print("set c r GET$/test-get-post.php?counter=1\r"); check();
24 Serial.print("set o f 1\r"); check();
25
26 // Calls open to launch the configured connection.
27 Serial.print("open\r"); check();
28 Serial.print("exit\r"); check();
29 }
```

Listing 3.1: Code to configure WiFi module using Raspberry Pi shield.

In this code we define how is going associate WiFi module to the WiFi AP. At the start defines what DHCP and TCP protocol is going to use and after sets which is the AP password and the WiFi ESSID (AP name) and joins. Next commands sets the DNS address and check that the association has been successful.

Figure 3.2: RN171 WiFi module.

The WiFi module is independent of the Waspmote, that is, when module switched on this configuration will be used to associate to the router and Waspmote will sent information using this connection. So this configuration only need to be run the first time.

3.2.4 System control

Up to now, we have a CommSensum little test environment prepared run, but we need too a computer to control what happens. The idea is connect a computer via Ethernet, and do a SSH connection to see server logs. At the same time this computer needs to have direct USB connection to the Waspmote.

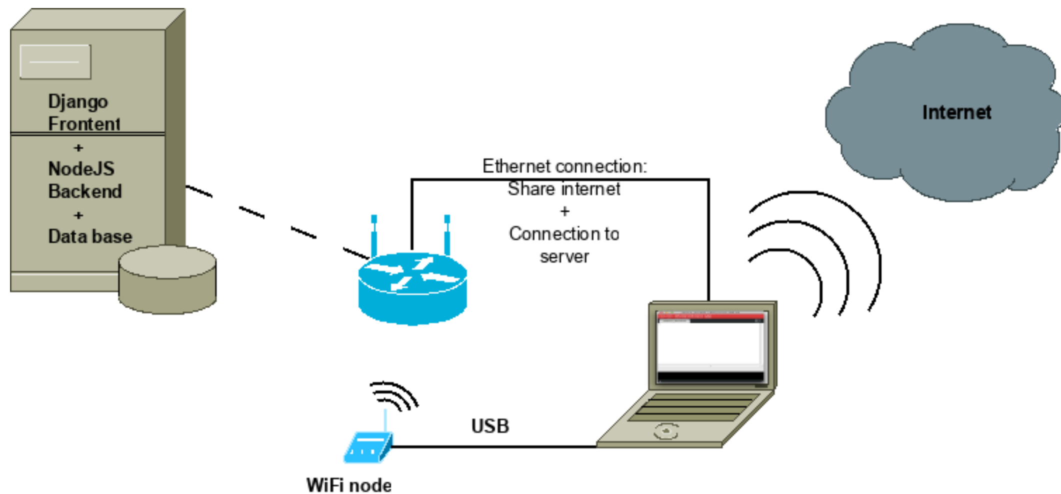


Figure 3.3: Test environment structure.

Using the connections we can see on the overview shown on Figure 3.3 we need to be able to access to different logs in order to check that all connections are done properly:

1. Frontend log: When we use front end page all changes will appear on this execution log.

2. TCPdump of waspmote IP: Using TCPdump we will see all message sent by Waspmote to the server. It is very useful to check that waspmote really send well all package around the network.

3. Api log: In this log we will check that message show on TCPdump is the same that it take NodeJS and the different changes done after.

4. Waspmote serial: Here we can check that nodes prepared package is well constructed and is well before send.

This system can be connected to the Internet sharing it by the computer to do that two connection need to be configured for that both connection one to the internet and the second to the test environment need to be done. After that you need to execute this bash script defined below and configure the firewall to share your internet connection. Do in this way, you can ensure that all the system is under control of the computer.

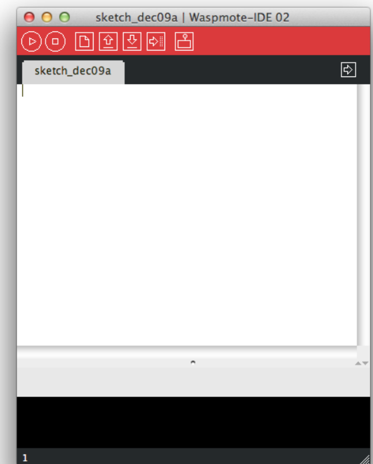
```
1  #!/bin/bash
2  #default example: sudo sh RouteToRaspberry.sh
3  #example with another gateway: sudo sh RouteToRaspberry.sh 192.168.1.1
4  
5  defGT=10.82.1.1
6  GT=${1:-$defGT}
7  
8  defServer=192.168.1.126
9  RASP=${2:-$defServer}
10 
```

```
11 echo "route -n"
12 route -n
13
14 echo "route del default"
15 route del default
16
17 echo "route add -net 0.0.0.0 gw $GT dev wlp3s0"
18 route add -net 0.0.0.0 gw $GT dev wlp3s0
19
20 echo "route -n"
21 route -n
22
23
24 echo "ssh -X pi@$RASP"
25 ssh -X pi@$RASP
```

Listing 3.2: Script to manage 2 connections in linux.

3.2.5 Waspmote IDE

For programming to Waspmote there is a specific IDE called Waspmote Pro IDE. This IDE, is based on Arduino IDE and uses *GCC* to compile. It allows you to compile and send compiled package to Waspmote. At the same time show serial information so you can see if on large period of time the code you implement continue running. To add new libraries to existing ones, it is necessary to go to the libraries folder and there for each new libraries file create new folder with the same name of the library you want to create.



This IDE is a open source, and you can download from: `http://www.libelium.com/development/waspmote/sdk_applications/`.

Chapter 4

Development and Results

Contents

4.1 Introduction **26**
4.2 AES implementation **26**
4.2.1 Libellium libraries 26
4.2.2 NodeJS libraries 27
4.2.3 Connection between NodeJS and Waspmote 27
4.3 RSA implementation **32**
4.3.1 Waspmote development 32
4.3.2 NodeJS development 35
4.4 Results **36**
4.4.1 Waspmote 36
4.4.2 Arduino Yun 37
4.4.3 More capacity hardware 37
4.5 Integration on CommSensum **37**
4.5.1 New Data Base tables 38
4.5.2 Changes on Frontend 38

4.1 Introduction

All work done looking for most common type of security, searching for related and similar work, analyzing and preparing test environment, is to reach to this chapter. Here we use all information we get to develop and implement a solution. The first step is develop AES encryption and make it work successfully between node and server. After that, RSA needs to be developed create public and private key connection and use AES encrypted aleatory session key to encrypt data. So if a session key is caught by intruder the system integrity is not in danger.

4.2 AES implementation

Communication between node and server without is done in raw text, this means that if we try to send encrypted bit-array unexpected thing can happens. For example, imagine that we encrypt data array and some bit take the form of *end-of-transmission character*. The message can not be totally received and the data will be lost.

Is for that the first step when you create an encrypted connection is define how send data. We analyze two format to send it Base64 and Hexadecimal string. To pass encrypted message to Base64 in C coding, different functions already exist, but all of this suppose a hardware consumption that we prefer to save. So the first step after defining Hexadecimal string to send data is to create a function c that covert uint8_t string to Hexadecimal char string output.

```
1 void toHex(uint8_t* data, uint16_t size, char* output){
2   uint16_t i;
3     for(i=0; i<size; i++){
4       sprintf(output+(i*2), "%02X", data[i]);
5     }
6 }
```

Listing 4.1: From uint8_t to hexadecimal char array function.

Here we can see the complexity of this function is not hard. We just use `sprintf` to create the Hex string in a loop of encrypted message size.

4.2.1 Libellium libraries

Libellium have WaspAES libraries to encrypt data using Rijandael algorithm. To use this the only need is use the function we can see below.

```
1      AES.encrypt(  key_size
2                  , key
3                  , content
4                  , encrypted_message
5                  , block_cypher
6                  , padding
7                  , IV);
```

Listing 4.2: AES encryption function.

Several variables need to be used, let's see them one by one:

key_size: Here need to define key size, can be 128, 192 or 256.

key: Here need to put the key is going to be use to encrypt.

content: The content that we want to encrypt

encrypted_message: this is the pointer with whom I will be targeted the encrypted message.

block_cypher: Only can be ECB or CBC

padding: Can be PKCS5, ZEROS

IV: Initialization Vector only need to be declared if you use ECB mode.

4.2.2 NodeJS libraries

NodeJS have integrated some crypto libraries that can encrypt and decrypt AES on different modes and key size. This libraries are base on OpenSSL application. Below we have an example of use.

```
1 var crypto = require('crypto'),
2 algorithm = 'aes-256-cbc',
3 password = 'd6F3Efeq';
4 function decrypt(text){
5 var decipher = crypto.createDecipher(algorithm,password)
6 var dec = decipher.update(text,'hex','utf8')
7 dec += decipher.final('utf8');
8 return dec;
9 }
```

Listing 4.3: Encrypt using NodeJS libraries.

This code is more intuitive than C, because you only need to change the algorithm that you want to use and execute decrypt function to get the decrypted data. We can use an hexadecimal representation of an string to decypher the message into "utf8", this means that we can use the message sent by the Waspmote directly.

4.2.3 Connection between NodeJS and Waspmote

So having libraries to encrypt in Waspmote and another libraries to decrypt in NodeJS, the next step should be unify the codes, encrypt, convert encrypted into hexadecimal char array and send it. But before, it is necessary to test if in both sides the encryption message it is the same. The problems start.

Connection problems

We assumed that both libraries will follow the AES standard but one does not. That is the reason that we try to simplify the maximum AES encryption to do test and identify easily the problem. This suppose to encrypt in 128bit, with no padding and putting messages that fill all encryption bytes. Even if, ECB mode is not recommended to these test was selected. The result of this, continue being two different encryption array.

Analyzing WaspAES function we notice that Waspmote puts "0" values at the end of password so, this too was changed to ensure that was not the cause of the problem. So as the error was not found, different encryption libraries [10] [3] [22] and applications [19] were used and retested trying to get the same result in both platform.

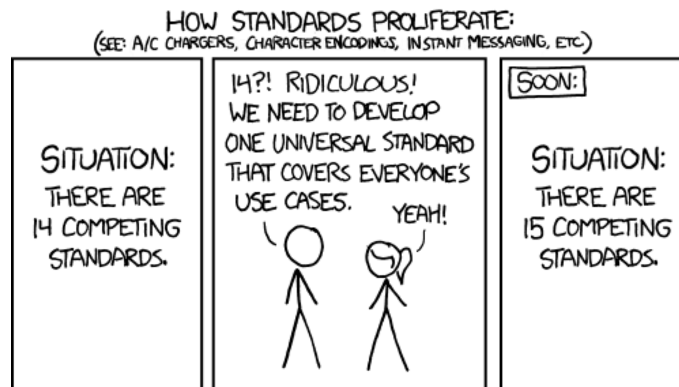


Figure 4.1: How standars proliferate.

After several proof I got to the root of the problem. Doing some test with OpenSSL [19] I realized that two modes were acepted. So encrypt a "hello00000000000"(zeros were used in order to eliminate padding) message using each node the result was different:

1. *echo 'hello00000000000' | openssl enc -aes-128-ecb -K 656c6f697469303030303030303030 -nosalt -out out.bin*

2. *echo 'hello00000000000' | openssl enc -aes-128-ecb -k eloiti0000000000 -nosalt -out out.bin*

Waspmote encrypt equal to first OpenSSL command and NodeJS equal to the second. The only different between them is how puts key and "-k" value that can be upper case or lower case. K upper case, it takes the password and encrypt using AES with the password, but k lowercase before encrypt derives the key to ensure that one intruder if get derive key can get the password. It is more secure, and many different platforms support this derive method but this is not in the AES standard.

So we need to get some libraries that allow to remove derive option or implement this derivation on the Waspmote. After search we get Node-Forge [17] encryption libraries for NodeJS that lets us choose if we want to derive the key or directly pass encryption key to AES.

Node-Forge

Node-Forge is a encryption libraries developed to NodeJ. Have a native implementation of TLS (and various other cryptography tools) in JavaScript. We need to have into account that, to ensure that the padding is the same on two sides need to be done manually.

```
1 var cipher = forge.cipher.createCipher('AES-CBC', key);
2 cipher.start({iv: iv});
3 cipher.update(forge.util.createBuffer(someBytes));
4 cipher.finish();
5 var encrypted = cipher.output;
6 // outputs encrypted hex
7 console.log(encrypted.toHex());
```

Listing 4.4: Encryption using Node-Forge.

If we look at the code, we realize that looks like other NodeJS encryption libraries or internet libraries but it have much more options. After some test, we got what we were looking, the same encryption in both sides and start the connection development.

Waspmote development

Once we have solved encryption problem it is the time to implement this encryption method on CommSensum code. All function to communicate to the API are on CSApi-Helper(CommSensum Api Helper) library. So to implement AES, the first step is load WaspAES libraries and do some definition on "CSApiHelper.h" file.

```
1  ...
2  #include <WaspAES.h>
3  ...
4  #define AES_PASSWORD "Pass we want"
5  ...
6
7
8  #define API_CONTENTTYPE "application/jwe"
9  ...
```

Listing 4.5: Needed definition in waspmote.

Here we can see what are the changes, fist of all is to include WaspAES libraries and after that it is necessary to define a password. To finish, we need to know that communication between nodes and server use Json communication but from now communications will be encrypted. So, different content type will be defined JSON Web Encryption(jwe).

So once we do definition part it is time to encrypt. We need to encrypt data before sending it so all code will be put in `sendHttpRequestSecureAESJsonArray()` function. After create content array to sent we will declare needed variables like `encrypted_length`, `encrypted_message` and `IV` to using these invoke AES function.

```
1  ...
2     buildCSObservationJSONArray(content, measures, num_measures);
3
```

```
4       // Variable for encrypted message's length
5       uint16_t encrypted_length;
6       // Declaration of variable encrypted message
7       uint8_t encrypted_message[1008];
8
9       //encrypted message
10      uint8_t IV[16] = { 0x00,0x01,0x02,0x03,...,0x0C,0x0D,0x0E,0x0F};
11      AES.encrypt(  128
12                  , AES_PASSWORD
13                  , content
14                  , encrypted_message
15                  , CBC
16                  , PKCS5
17                  , IV);
18      encrypted_length = AES.sizeOfBlocks(content);
```

Listing 4.6: Encrypt using AES in waspmote.

After that all headers are sent but before sending our encrypted message it is necessary to convert it on hexadecimal array. So, toHex function is called and the message is sent using WIFI.send() function.

```
1       char output[encrypted_length*2];
2       toHex(encrypted_message, encrypted_length, output);
3
4       sprintf(tmp, "Content-Length: %d\r\n", strlen(output));
5       WIFI.send(tmp);
6
7       WIFI.send(F("Connection: close\r\n\r\n"));
8       WIFI.send(output);
9       free(output);
10      free(encrypted_message);//free encrypted
11      free(IV);//free iv
12      free(content);
13      free(tmp);
14      content=NULL;
15      tmp=NULL;
16      //wait for the HTTP Reply and check it's an 200 OK
17  ...
```

Listing 4.7: Send AES encrypted data in waspmote.

At the end of the code all declared pointers are released. We need to realize that, this code will be put on two functions because each node can execute this function to send an array of measures or by means of another functions send just one measure. Encryption possibility need to be defined in both functions.

NodeJS development

To implement AES some changes was done on functions.js file, first of all, "jwe" support needs to be added. There are two functions to get data `gatParams` and `getParamsArray`. In both, decryption function will be define.

- `gatParams`: Gets a unique param from node.
- `getParamsArray`: Gets a group of params arrived in array.

So in both function when content-type is defined we need to add support to "jwe" adding code below.

```
 ...
else if(
    ('content-type' in req.headers) &&
    (req.headers['content-type'].toLowerCase().indexOf("jwe")
    != -1))
      {
      ...
```

Listing 4.8: Encrypt using RSA function in waspmote.

An AES decryption function needs to be defined, this function, supports ECB and CBC modes only with IV definition. That is, if IV is not defined (IV=null), ECB mode will be used. The different between ECB and CBC is that in ECB is the same cipher block mode CBC mode but with 0 array IV. So we take Node-Forge libraries and we define this function.

```
var decryptAES = function(data, key, iv){
var forge = require('node-forge');
// Note: a key size of 16 bytes will use AES-128,
// 24 => AES-192, 32 => AES-256
var decipher = forge.cipher.createDecipher('AES-CBC', key);
//If iv = null is a ecb mode that is a 0 array iv
if(iv==null){
   var buffer = forge.util.createBuffer();
   buffer.fillWithByte(0,15);
   iv = buffer.getBytes(16);
}
decipher.start({iv: iv});
decipher.update(forge.util.createBuffer(
    forge.util.binary.hex.decode(data)));
decipher.finish();
return decipher.output.toString();
}
```

Listing 4.9: Decript AES function on NodeJS.

To run properly before execute this function a "`switch-case`" needs to be created to know what type of encryption user data that the Waspmote sends. This information will be obtained from data base using what is the node is sending data.

4.3 RSA implementation

To implement RSA on CommSensum there is no need to use, on server side, different libraries. Like Node-Forge libraries supports RSA and give us freedom to manage encryption padding the same libraries have been used. On node side, Libellium provides WaspRSA libraries has been used.

4.3.1 Waspmote development

Like in AES implementation, the first step to encrypt in Waspmote, is defining some libraries on *CSApiHelper.h*. As we are using public-private key encryption system some values need to be declared too.

Like we explain on chapter 2, to encrypt and ensure user identity in RSA two pair of key are needed. For this, in node side of RSA implementation, server public key and nodes private key are needed.

```
1   ...
2   #include <WaspRSA.h>
3   ...
4   #define modulus "F12C7B9415B88146B...6364D07834E524F271A23CB576D"
5   #define public_exponent "00010001"
6   #define public_modulus_sign "ACA22E21A4...D620D5CEE16311B8417A9D"
7   #define private_exponent_sign "68b703768298d5433c1b3...82d44d63d"
8   ...
9   #define API_CONTENTTYPE "application/jwe"
10  ...
```

Listing 4.10: Necesary defines on CSApiHelper.

Like Waspmote is an embedded system it cannot use keys directly. It needs some key parts that can be derived from PEM [21] type key format. These parts are modulus and exponent.

So, different parts will need to create RSA security, random password generation, encryption of this password and a digital signature of sent message.

```
1 void gen_random(char *s,int len) {
2    static const char alphanum[] = "0123456789ABCDEF
3            GHIJKLMNOPQRSTUVWXYZ";
4    int i;
5    for (i = 0; i < len; ++i) {
6        s[i] = alphanum[rand() % (sizeof(alphanum) - 1)];
7    }
8    s[len] = 0;
9 }
```

Listing 4.11: Random password generate function.

This first function, “gen_random” only takes and alphanumeric array and takes random characters from them to create a random password.

So once we have a random session key (password), using server public key we encrypt this password using WaspRSA libraries, so only the server can decrypt it.

```
void cryptRSA(char * password, char * session_enc,
        uint8_t pass_size, uint8_t enc_size){
    gen_random(password, pass_size);
    char* hexPass;
    hexPass=(char*)calloc((pass_size*2)+1,sizeof(char));
    toHex(password, pass_size*2, hexPass);
  RSA.encrypt(hexPass
    , public_exponent
    , modulus
    , session_enc
    , 300);
        free(hexPass);
        hexPass=NULL;
}
```

Listing 4.12: Encrypt using RSA function in waspmote.

Only one step more is needed, ensure that each node identity cannot be supplanted. For that, digital signature needs to be calculated and sent to server. This function takes the messages and calculates it's HASH. After that encrypts this HASH using RSA with node private key like we explain in chapter 2.

```
void digSign(char* message, char* dig_sign, uint8_t length){
  uint8_t hash_message_sha1[20];
  HASH.sha( SHA1, hash_message_sha1, (uint8_t*)message,
      strlen(message)*8);
  RSA.encrypt(hash_message_sha1
    , 20
    , private_exponent_sign
    , public_modulus_sign
    , dig_sign
    , length);
}
```

Listing 4.13: Waspmote digital signature function.

So, this function returns the digital signature of the message we want to send. We need to take into account that digital signature is not supported by Libellium, but showing RSA formula we realize that is the same formulation of encrypting with public key but using private exponent and modulus. The following equation 4.1 is the RSA encryption function where "c ēncrypted message", "m m̄essage", "e p̄ublic exponent" and "n = modulus".

$$c = m^e (mod\ n) \tag{4.1}$$

To decrypt message we can see that the formula is the same but where "m d̄ecrypted message", "c ēncrypted message", "dp̄rivate exponent" and "n = modulus". So the only

difference is used exponent, public to cipher and private to decipher. See this change on formula 4.2.

$$m = c^d (mod\ n) \tag{4.2}$$

After programming this part of the code, we realize that if Libelium does not support digital signature is because Waspmote have not enough SRAM memory to run it. But this does not mean this code is not useful because as it is C code it is possible to run on different Arduino models (or future Waspmote). So we finish this coding.

With each part programmed, the only missing code is how to relate each part. On following Listing ?? code we can see how, `cryptRSA` create session key and session encrypted. After that, digital signature is created. So in this part, all RSA functions are executed. After that using AES cryptography encrypts data array and as in previous AES functionality pass this encrypted data to hexadecimal array.

```
1      char* session;
2      session=(char*)calloc(32,sizeof(char));
3      char session_enc[300];
4      cryptRSA(session,session_enc,32,300);
5      free(session);
6      char* dig_sign;
7      dig_sign=(char*)calloc(257,sizeof(char));
8      digSign(content,dig_sign, 257);
9
10     // Variable for encrypted message's length gero bidaltzeko
11     uint16_t encrypted_length;
12     // Declaration of variable encrypted message
13     uint8_t encrypted_message[1008];
14     //encrypted message
15     uint8_t IV[16] = { 0x00,0x01,0x02,0x03,...,0x0C,0x0D,0x0E,0x0F};
16     AES.encrypt(  128
17                 , session
18                 , content
19                 , encrypted_message
20                 , CBC
21                 , PKCS5
22                 , IV);
23     encrypted_length = AES.sizeOfBlocks(content);
24     free(content);
25
26     char* output;
27     output=(char*)calloc(encrypted_length*2,sizeof(char));
28     toHex(encrypted_message, encrypted_length, output);
29     free(encrypted_message);//free encrypted
```

Listing 4.14: Encrypting data array using RSA in waspmote.

With RSA encrypted data, new headers are needed "X-ApiNode" and "X-Digital-Signature". The node id is required to tell server what node is and look into the data base

what type of encryption use each one. If we use RSA a digital signature also to be sent to let the server know that the node is really who it says it is.

```
1      ...
2      sprintf(tmp, "X-ApiNode: %i\r\n",API_NODEID);
3      WIFI.send(tmp);
4      //WIFI.send(F("X-ApiNode: "       API_NODEID"\r\n"));
5      sprintf(tmp, "X-SessionKey: %s\r\n",session_enc);
6      WIFI.send(tmp);
7      sprintf(tmp, "X-Digital-Signature: %s\r\n",dig_sign);
8      WIFI.send(tmp);
9      sprintf(tmp, "Content-Length: %d\r\n", strlen(output));
10     WIFI.send(tmp);
11     WIFI.send(F("Connection: close\r\n\r\n"));
```

Listing 4.15: Send RSA data from waspmote.

4.3.2 NodeJS development

With the implementation done before for encryption packages to provide of RSA, it is easier to implement it. On the encryption switch case, only it is necessary to add RSA-1024 and execute the following function.

```
1  var decryptRSA = function(data,
2                            sessionKey,
3                            pem,
4                            dig_sig,
5                            user_public,
6                            iv)
7  {
8      var forge = require('node-forge');
9      var privateKey = forge.pki.privateKeyFromPem(pem);
10     var decryptedKey = privateKey.decrypt(
11       forge.util.createBuffer(
12   forge.util.binary.hex.decode(sessionKey)).getBytes(),"NONE");
13 
14     //Because node-forge and waspmote use different
15     //padding we need to remove zeros manually.
16     var key = forge.util.createBuffer(decryptedKey).toHex();
17     key = key.replace(/^0+/, '');
18 
19     //Package decryption function
20     var decrypted = decryptAES(data,forge.util.createBuffer(
21   forge.util.binary.hex.decode(key)).data,iv);
22     var md = forge.md.sha1.create();
23     md.update(decrypted);
24     var publicKey = forge.pki.publicKeyFromPem(user_public);
25     var sha1 = forge.pki.rsa.decrypt(forge.util.createBuffer(
```

```
26    forge.util.binary.hex.decode(dig_sig)).getBytes(),
27    publicKey, true, false);
28      key = sha1.toHex();
29      key = key.replace(/^0+/, '');
30      console.log("Signature: "+key)
31      if(md.digest().toHex()===key){
32        console.log("Signature verified.");
33        return decrypted;
34      }
35      else{
36        console.log("Signature not verified.");
37          return null;
38      }
39  }
```

Listing 4.16: decryptRSA function.

In this function enter the following parameters: encrypted `data`, `sessionKey`, server `pem`, digital signature (`dig_sig`), node public PEM (`user_public`) and AES IV (null if use ECB).

So first of all decrypt RSA encrypted password and removes zeros manually because node-forge and Waspmote use different padding. After that, decrypt data package and calculates data array HASH. To ensure that this node is you who claims to be decrypt digital signature and compares if digital signature content is equal to HASH. If the signature is verified returns data to save at database. If not verified sends null value and no data is saved.

4.4 Results

In all development part we create different security levels, with different security types adapted to each board. It is impossible to run HTTPS on motes for now, but we can be prepared to near future hardware. Development code results are distributed on different used security types.

4.4.1 Waspmote

After different test, with different code and reducing SRAM consumption to the minimum in cryptography function I have found that there is no possibility to run RSA libraries in Waspmote. Waspmote has the capability to encrypt with public modulus using 3.6KB of SRAM but when we try to do a digital signature, this SRAM consumption grows to more than the 8KB. Even if we use the simplest program in the Waspmote the SRAM free after putting all other functionality is roughly 1.5KB. Away even 3.6KB of encryption with public modulus.

For now, the only cipher we can use on motes hardware like Waspmote is different AES modes with the risk that entails. Always looking to the future hardware that is possible to have new low consumption hardware (new WiFi modules) that allows HTTPS connection.

4.4.2 Arduino Yun

It is not possible to run RSA code in Waspmote but there are Arduino boards that have more SRAM than Waspmote. One of this boards is Ardunio yun. Arduino yun has two parts, a microcontroller board based on the ATmega32u4 and the Atheros AR9331. On figure 4.2 we can see how they communicate both processors. We can see how to send data ATmega32u4 processor need to use Atheros part. So Atheros part have enough hardware characteristic to run RSA code, it is possible to run this code there.

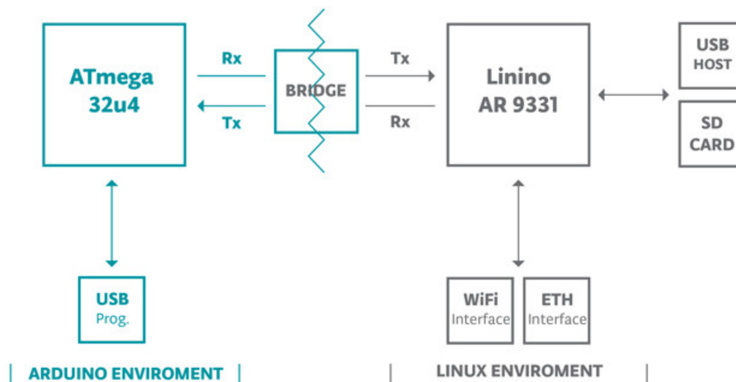


Figure 4.2: Arduino yun architecture.

We need to take into account, that this is possible only because Arduino Yun is not a low power consumption mote. So it is not an option to put in a forest for example.

4.4.3 More capacity hardware

Now there are more boards with different capabilities like the like Raspberry Pi that can create HTTPS connection without problems. But, in contrast motes, Raspberry Pi is more like a computer so, it have a much higher consumption. It is very possible too, that RN171 successor have native HTTPS so, with this type of technology it is very possible to get in few time motes with HTTPS security.

4.5 Integration on CommSensum

We mention before some changes to run all written code on CommSensum API server, but still are missing changes. But we do not explain how changes are need it on the data base to support security. On frontend server too, changes are needed to allow new users to define what type of security they want. This changes need to be saved on the database in order to allow the API server to get this data.

4.5.1 New Data Base tables

So to provide a database that can use different types of encryption some tables need to be created linked to existing ones. On figure 4.3 we have the tables and what data they save. Nodes table already exist on CommSensum, all others tables have been added to provide a possibility to add new cryptography types only adding new tables, without changes. The relations means, that each node have a unique cryptography type. Each

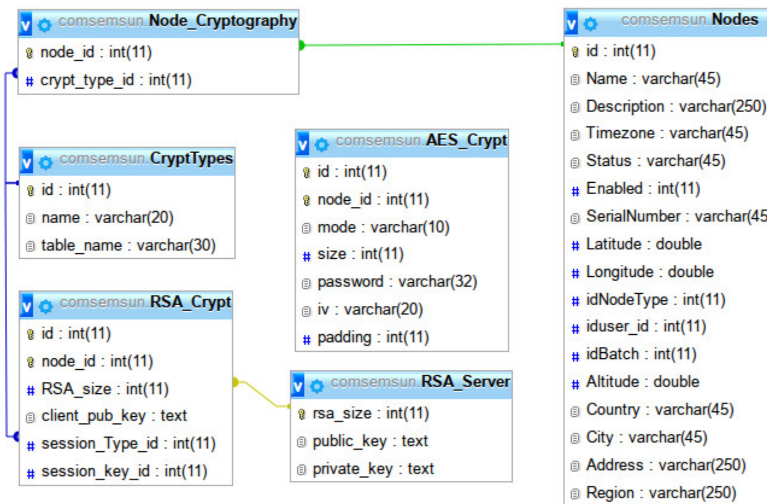


Figure 4.3: New data base tables.

cryptography type will have its unique table where it will save the node data. So if we want to implement DES we only need to define new cryptography and create a new table with data that needs this cryptography type.

We can see too, how RSA has a relation with cryptTypes table, this is because RSA to encrypt data use another cipher, so, need to be flexible we add this relation and this information will keep on this type table.

4.5.2 Changes on Frontend

On frontend, changes are needed to provide users adding to nodes security type. So, in existing forms new inputs have to appear and when the form is saved execute insert queries defined before on database.

Chapter 5

Conclusions and Future work

Contents

5.1 Conclusions **40**
5.2 Future work **41**

5.1 Conclusions

First thing we want to highlight is that on this thesis we provide a solution that works. This project, put the first step to provide security in community sensor networks although we know that is not the ideal solution.

In computing it is really important to follow the standards step by step if any exists. Because if anybody wants to continue or extend any work that does not follow the standard he can lose a lot of time or do not get anything from this work. Then, an important lesson is that you should always follow the standard.

Apart from following standards, other important thing is to have a good documentation. A lot of problems can be avoided if you have information of it. During the project there have been several problems related with documentation, for example, NodeJS has no documentation explaining how you have to program, only have some examples. Other example is Libelium that ensures the availability of a HTTPS connection using WiFi module but does not specify how they implement it (using Meshlium.)

As we mention during the entire document, Crowdsourcing Platform topic will be more important in the coming years and security of this networks will be crucial. It is why even if there is a lot of work done on embed security, there is no form to implement something as safe as HTTPS on a independent mote for home. So there is still important work to be done in this topic.

As a more specific conclusion for this project, when you work with security you need to take into account that you can implement ciphers in all part of them. You need to analyze all the system and provide security on all routes that can have a weakness. But, we have to take into account what we are ciphering and where since power consumption might increase.

5.2 Future work

For security topic a lot of work needs to be done, but the first step is to implement HTTPS protocol on the server Frontend side. So doing this, all user can create new nodes securely. At the same time, different security levels need to be defined on the server and create different levels and permissions.

We mention during all the thesis, the possibility of use a new generation WiFi module, the next of RN171, can have HTTPS connection. Taking into account that, this type of modules already exist [16]. The problem of this module is that is not assembled like a functional module (you need to solder manually.) So, there are two possibilities, wait the development of this new module or program RN171 to create HTTPS connection. For that, wide analysis of RN171 and its software need to be done and then define if it is possible because RN171 might have enough hardware requisites to run it.

Looking at the future new security types will be developed, security sent by better hardware, but we need to have into account that to this time some nodes will be already deployed. So when new cryptography type has been added the cryptography done before should have support.

References

[1] Aire limpio ya. http://microdonaciones.hazloposible.org/proyectos/detalle/?idProyecto=112.

[2] Arduino project webpage. http://www.arduino.cc/.

[3] AVR-crypto-lib C language libraries for cryptography. https://www.das-labor.org/wiki/AVR-Crypto-Lib/en.

[4] Ciudadanas y ciudadanos por el cambio en Leganes. http://www.ciudadanosporelcambio.com/dblog/articolo.asp?articolo=175.

[5] CompNet Research Group. http://compnet.ac.upc.edu/intranet/.

[6] Contaminación en Madrid: Gallardón y el Ayuntamiento mienten. http://www.ecologistasenaccion.org/article731.html.

[7] Django architecture explanation. https://docs.djangoproject.com/en/dev/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-a

[8] Django project webpage. https://www.djangoproject.com/.

[9] Facultat d'Informatica de Barcelona (FIB). http://www.fib.upc.edu/fib.html.

[10] Google provides JavaScript cryption libraries. https://code.google.com/p/crypto-js/.

[11] JavaScript information webpage. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[12] La calidad del aire en Madrid. http://elpais.com/diario/2003/05/05/madrid/1052133871_850215.html.

[13] Libellium case of study webpage. http://www.libelium.com/case-studies/.

[14] Libellium company webpage. http://www.libelium.com/company/.

[15] Libellium WiFi Module Networking Guide. http://www.libelium.com/development/waspmote/documentation/WiFi-networking-guide/.

[16] Low-power WiFi module with HTTPS connection. http://www.gainspan.com/gs1500m.

[17] Node-Forge encryption JavaScript libraries for NodeJS. https://www.npmjs.com/package/node-forge.

[18] NodeJS organization webpage. http://nodejs.org/.

[19] OpenSSL open source toolkit for SSL. http://www.openssl.org.

[20] PeerSec Networks, MatrixSSL. http://www.matrixssl.org.

[21] PEM format defined by OpenSSL. https://www.openssl.org/docs/apps/rsa.html.

[22] PolarSSL C language libraries for cryptography. https://polarssl.org/.

[23] RN-171 microchip product web page. http://www.microchip.com/wwwproducts/Devices.aspx?product=RN171.

[24] RN-171 microchip programming guide. http://ww1.microchip.com/downloads/en/DeviceDoc/50002230A.pdf.

[25] Roving Networks RN-171 datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/70005171A.pdf.

[26] The CommSensum Platform overview. http://compnet.ac.upc.edu/intranet/?q=node/199.

[27] The CommSensum webpage. http://commsensum.pc.ac.upc.edu/.

[28] Waspmote Libellium libraries. https://github.com/Libelium/waspmoteapi.

[29] Waspmote web page. http://www.libelium.com/products/waspmote/hardware/.

[30] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[31] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1998.

[32] Vipul Gupta, Michael Wurm, Yu Zhu, Matthew Millard, Stephen Fung, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded internet. *Pervasive and Mobile Computing*, 1(4):425–445, 2005.

[33] Woo-Young Jung. Ssnail: security of sensor networks for all-ip world.

[34] Wooyoung Jung, Sungmin Hong, Minkeun Ha, Young-Joo Kim, and Daeyoung Kim. Ssl-based lightweight security of ip-based wireless sensor networks. In *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on*, pages 1112–1117. IEEE, 2009.

[35] Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010.

[36] Sara Robinson. Still guarding secrets after years of attacks, rsa earns accolades for its founders. *SIAM News*, 36(5):1–4, 2003.

[37] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson, Tadayoshi Kohno, and Mike Stay. The twofish team's final comments on aes selection. *AES round*, 2, 2000.