Implementation and Evaluation of BSD Elliptic Curve Cryptography

Oriol Piñol Piñol

Master of Science Thesis

Software Engineering in Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology

SICS Swedish Institute of Computer Science
CSL Computer Systems Laboratory
Stockholm, Sweden

04 November 2014

TRITA-ICT-EX-2014:165

Examiner: Dr. Jim Dowling
Supervisors: Dr. Shahid Raza and Joakim Eriksson
Abstract

Security is recently arising as an important issue for the Internet of Things (IoT). Efficient ways to provide secure communication between devices and sensors is crucial for the IoT devices, which are becoming more and more used and spread in a variety of fields. In this context, Elliptic Curve Cryptography (ECC) is considered as a strong candidate to provide security while being able to be functional in an environment with strong requirements and limitations such as wireless sensor networks (WSN). The solutions used need to be efficient for devices that have some important restrictions on memory availability and battery life. In this master thesis we present a lightweight BSD-based implementation of the Elliptic Curve Cryptography (ECC) for the Contiki OS and its evaluation. We show the feasibility of the implementation and use of this cryptography in the IoT by a thorough evaluation of the solution by analyzing the performance using different implementations and optimizations of the used algorithms, and also by evaluating it in a real hardware environment. The evaluation of ECC shows that it can adapt to the upcoming challenges, thanks to the level of security that it provides with a smaller size of keys when compared to other legacy cryptography schemes.
Acknowledgements

I would like to thank the Networked Embedded Systems group at SICS for all the support and the help during my work there. Especially I would like to thank my supervisors Shahid Raza and Joakim Eriksson for their help and guidance during this time. I would also like to thank the people at Yanzi Networks for the opportunity to test my work on their system and the doors that this opened afterwards. Finally I would like to thank my family and my friends for the support during the realization of this Master Thesis work.
# Contents

## 1 Introduction

1.1 Background .............................................. 1  
1.2 Problem description ......................................... 2  
1.3 Purpose .................................................. 2  
1.4 Goal, Benefits, Ethics and Sustainability ....................... 2  
1.5 Methodology / Methods ...................................... 3  
1.6 Delimitations ............................................ 3  
1.7 Outline ................................................ 3  

## 2 Background

2.1 Public Key Cryptography .................................... 5  
   2.1.1 RSA .................................................. 6  
   2.1.2 ECC ................................................. 7  
   2.1.3 Diffie-Hellman key exchange protocol ....................... 8  
2.2 Elliptic Curve Cryptography ................................ 9  
   2.2.1 Mathematical foundations ................................. 9  
   2.2.2 Operations ........................................... 11  
      2.2.2.1 Representation and coordinate systems .............. 11  
      2.2.2.2 Point addition .................................. 12  
      2.2.2.3 Scalar multiplication ............................. 13  
      2.2.2.4 Key generation .................................. 13  
      2.2.2.5 Elliptic Curve Diffie Hellman (ECDH) ............... 13  
      2.2.2.6 Elliptic Curve Digital Signature Algorithm (ECDSA) 14  
   2.2.3 Comparison/Advantages ................................ 14  
2.3 Arbitrary precision arithmetic ................................ 15  
   2.3.1 Basic operations ..................................... 16  
      2.3.1.1 Multiplication and squaring ....................... 16  
      2.3.1.2 Division ....................................... 16  
      2.3.1.3 Special operations .............................. 17  
2.4 Contiki ................................................. 17  
   2.4.1 Protothreads ........................................ 17
## Contents

2.4.2 Code, compile and upload Contiki .................................. 18
2.5 Related work ............................................................. 20

### 3 Design and implementation

3.1 Design ................................................................. 21
3.2 Implementation ........................................................ 22
3.2.1 BigInteger library implementation .............................. 22
3.2.1.1 Basic operations .............................................. 23
3.2.1.2 Modular arithmetic .......................................... 25
3.2.1.3 Manipulation algorithms ................................... 26
3.2.2 Elliptic Curve Cryptography library implementation ...... 26
3.2.2.1 EC point addition ............................................ 27
3.2.2.2 Scalar multiplication ........................................ 28
3.2.2.3 Key pair generation .......................................... 28
3.2.2.4 EC Diffie-Hellman exchange ................................ 29
3.2.2.5 ECDSA: Elliptic Curve Digital Signature Algorithm ..... 29

### 4 Evaluation

4.1 Evaluation set-up ...................................................... 31
4.2 Evaluation parameters and configurations .......................... 31
4.3 ECC operations evaluation ............................................ 32
4.3.1 Key pair generation .............................................. 33
4.3.2 EC Diffie-Hellman exchange .................................... 35
4.3.3 ECDSA .............................................................. 37
4.3.4 Memory usage ..................................................... 38
4.4 Energy consumption .................................................. 40
4.5 Conclusions ............................................................ 41

### 5 Conclusions

5.1 Conclusion ............................................................ 43
5.2 Future work .......................................................... 43

Bibliography ............................................................... 45

A How to run ............................................................... 49
List of Figures

2.1 Representation of two different elliptic curves over $\mathbb{R}$, with positive (a) and negative (b) discriminant . . . . . . . . . . . . . . 10

3.1 Libraries files’ relationship. . . . . . . . . . . . . . . . 22

4.1 EC key pair generation computational time. . . . . . . . . . . . . . . 34
4.2 ECDH key exchange latency. . . . . . . . . . . . . . . . . . . . . 36
4.3 ECDH shared secret generation computational time. . . . . . . . . 37
4.4 ECDSA signature generation and verification computational time. 38
4.5 EC key-pair generation and ECDH exchange computational time. 42
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Key size comparison RSA/ECC for equivalent security level</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Cost in operations of point addition and point doubling for different coordinate systems.</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Flash memory usage in bytes for different key sizes</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>RAM stack memory usage in bytes in function of the key size and coordinate system.</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Energy consumption in mJ for the main ECC operations in jacobian coordinate system with a key size of 256 bits.</td>
<td>41</td>
</tr>
</tbody>
</table>
List of Algorithms

1. ECDSA Signature generation algorithm . . . . . . . . . . . . . . . 30
2. ECDSA Signature verification algorithm . . . . . . . . . . . . . . 30
## List of Code blocks

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>C preprocessor implementation of Contiki OS protothread macros and implementation of local continuations.</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>C implementation of local continuations using switch statement.</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Example code of the Hello World application included in Contiki</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Implementation of the calculation of the division reciprocal.</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Implementation of the double-precision word by single-precision word using division reciprocal</td>
<td>24</td>
</tr>
</tbody>
</table>
## List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest, Shamir, Adleman</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>DH</td>
<td>Diffie-Hellman</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>HoAC</td>
<td>Handbook of Applied Cryptography</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

We are moving towards a world where all kind of objects will be connected and Machine-to-machine (M2M) communication will become a huge part of the communications. This is commonly called the Internet of Things (IoT), one of its definitions is “a vision where objects become part of the Internet: where every object is uniquely identified, and accessible to the network, its position and status known, where services and intelligence are added to this expanded Internet” in [1]. These devices have important constraints regarding power, memory and processing power compared to other devices and, therefore, in many cases, there is a need to reconsider the existing solutions for these devices’ operations and communication. Communication protocols have been twisted to adapt to the mentioned constraints and improve the efficiency in this kind of devices [2] [3], and even special OS, such as the Contiki OS [4], have been designed especially to adapt to this new paradigm [5]. Security is a big issue in this new environment. Privacy, Identity management, security and access control are an important challenge [1] and also the issue of how to assign credentials to distinguish access to certain information by the machines, which might not be related to a person anymore, arises. Protocols to establish security in the communications between devices such as those belonging to wireless sensor networks (WSN) have been implemented focusing in power efficiency and enabling communication with other devices over the Internet [2]. In this paradigm, public-key cryptography has been shown to be a very viable alternative [6]. For many applications in this area, energy is considered the main restriction but public key cryptography has proven being able to fulfill these requirements and be able to perform in such restrained environments.
1.2 Problem description

The protocols and applications used to provide secure communication between devices nowadays are implemented for machines that are not limited by their power or memory in a drastic way. On the other hand, the Internet of Things is mainly composed by devices that usually have important power restrictions. Therefore, power-efficient, lightweight protocols need to be designed. Since security is such an important issue the security protocols need to be also adapted to this paradigm. Nowadays public key cryptography with TLS and RSA are the de-facto standards in secure communications but they do not fit the aforementioned need. Protocols such as Datagram Transport Layer Security (DTLS) [7] have been design to approach this problem. But on the application level security, Elliptic Curve Cryptography (ECC) shows up as an alternative to RSA. Elliptic curves achieve the same level of security as RSA algorithms with less number of bits, implying faster, and less power-consuming schemes providing the same security [8], [9].

This work tries to solve the need for a lightweight design of ECC that is suitable for the IoT and constrained devices, particularly in the Contiki OS and using a BSD license.

1.3 Purpose

The purpose of this thesis report is to present the design, implementation and evaluation of an ECC lightweight design that provides a good performance in constrained devices in the IoT context, in particular in the Contiki OS.

1.4 Goal, Benefits, Ethics and Sustainability

The goal of this project is to obtain a BSD lightweight ECC implementation suitable for constraint devices in Contiki OS and perform an evaluation of the implementation in typical hardware used in Wireless Sensor Networks. The reference that will be followed in this work and the goal is to correspond to the specifications in the Request For Comments (RFC) 6090 [10].

To achieve this goal, the following tasks were performed:

- Creation of a lightweight design of ECC for the IoT.
- Implementation of ECC in Constraint Devices in Contiki OS.
- Thorough evaluation of ECC on real hardware and in a real IoT environment from Yanzi Networks.
1.5 **Methodology / Methods**

This project had three clearly differentiated parts that needed different approaches and methodologies to be used. The first part consisted in a literature study to understand the background and to propose and design the most appropriate solution for the posed problem. After that, the design was implemented following an empirical research method. From the hypotheses and the design that came out of the data collection and the following analysis, the corresponding implementation was realized. Finally, we performed an evaluation of the implementation using a quantitative approach. We obtained experimental performance results, analyzed them and used a deductive approach to draw conclusions regarding the performance of the design and the implementation of the work.

1.6 **Delimitations**

The implementation of this master thesis is specific for the Contiki OS for Wireless Sensor Networks. Although this is a very extended Operating System for embedded systems, the design and implementation is specific for it. The evaluation is performed on the STM32W platform and it is specific for this platform as it was the platform used during development and, after, evaluation on a real use case in the nodes from Yanzi Networks AB. The behavior and performance on other platforms could differ considerably from the results presented in this work.

1.7 **Outline**

Chapter 2 will provide the background related to this master thesis and help the reader understand the area of this thesis and the rest of this work. After that, chapter 3 describes the process of the design and the implementation of this work. The design of the architecture of this work is presented and we explain how it was implemented to achieve an output satisfying the design. Then, in chapter 4, a thorough evaluation of the work is provided. The test set-up and the different parameters for evaluation are exposed as well as the results of this evaluation. Finally, in chapter 5, the conclusions obtained in this work are presented to summarize the results and the output of this work as well as suggest future work in the area. To complement the presented work, in the Appendix can be found a quick guide on how to use the library in the Contiki OS.
Chapter 2

Background

2.1 Public Key Cryptography

A Public-key cryptosystem, as defined by Diffie and Hellman in New Directions in Cryptography [11], is a pair of families \( \{ E_K \}_{K \in \mathcal{K}} \) and \( \{ D_K \}_{K \in \mathcal{K}} \) of algorithms representing invertible transformations,

\[
E_K : \{ M \} \rightarrow \{ M \}
\]

\[
D_K : \{ M \} \rightarrow \{ M \}
\]

on a finite message space \( \{ M \} \), such that

1. for every \( K \in \mathcal{K} \), \( E_K \) is the inverse of \( D_K \),

2. for every \( K \in \mathcal{K} \) and \( M \in \{ M \} \), the algorithms \( E_K \) and \( D_K \) are easy to compute,

3. for almost every \( K \in \mathcal{K} \), each easily computed algorithm equivalent to \( D_K \) is computationally infeasible to derive from \( E_K \),

4. for every \( K \in \mathcal{K} \), it is feasible to compute inverse pairs \( E_K \) and \( D_K \) from \( K \).

This gives public key cryptography some interesting properties.

As stated in the Handbook of Applied (HoAC) [12] ”The concept of public-key encryption is simple and elegant, but has far-reaching consequences”. For example, and in opposition to symmetric-key cryptography, in the public key scheme there is no need to transmit an encryption key using a secure channel to establish a secure communication channel. The encryption key can be transmitted using an unsecured channel (which can be the same that will transport the
ciphertext) without compromising the security of the system due to the third property exposed, since it is computationally infeasible to derive the decryption key from the information made public. Also, there is no need to create a different key for communicating with different entities. The properties of the public-key cryptography allows the use of the same key pair to communicate with any other entity.

This scheme presents, though, an important issue: It creates the necessity of authentication. Impersonation of the receiver arises as one of the main issues for the scheme since a potential attacker can provide the sender a public encryption key \( E' \) that the sender assumes (incorrectly) belonging to the intended receiver. This way, the attacker can intercept the messages and decrypt them.

It has been shown that generally public key cryptosystems are slower than the securely equivalent symmetric systems. Mainly for this reason, it is of common practice to use public key cryptography to establish a shared key for a symmetric cryptosystem to be later used for the data communication between the two different parties.

Following we present two different public-key schemes, RSA and ECC. These two public-key cryptosystems have been considered viable for Wireless Sensor Networks [8], which are the main focus of this work.

### 2.1.1 RSA

RSA stands for Rivest, Shamir and Adleman, who first publicly described this public-key cryptosystem back in 1977 and published one year later in the publication: A Method for Obtaining Digital Signatures and Public-key Cryptosystems [13]. The RSA algorithm relies in the difficulty to solve the RSA problem (RSAP), introduced in the mentioned Rivest, Shamir and Adleman paper, and that is based on the computational difficulty and effort to factorize large numbers.

Rivest, Shamir and Adleman suggested the following method to encrypt and decrypt messages using a public encryption key \((e, n)\) and a private key \((d, n)\) where all \(e, d,\) and \(n\) are positive integers.

\[
C \equiv E(M) \equiv M^e \pmod{n}, \text{ for a message } M.
\]

\[
D(C) \equiv C^d \pmod{n}, \text{ for a ciphertext } C.
\]

To generate the keys, first 2 big primes are selected to generate \( n = p \times q \). After that, \(d\) is selected as a large random prime which is relatively prime to \(p\). That is:

\[
\gcd(d, (p - 1) \times (q - 1)) = 1
\]
Finally $e$ is calculated as the multiplicative inverse of $d$ modulo $(p - 1) \times (q - 1)$. That is:

$$e \times d \equiv 1 \mod ((p - 1) \times (q - 1)).$$

In their paper, the authors show that the obvious approaches to break the system are at least as difficult as factoring $n$, and that there is no efficient algorithm known for solving this problem.

RSA is the most widely used public-key algorithm but due to the fact that it relies on the computational effort required by the factorization of big prime numbers, it requires keys longer keys as the computational power and resources increase. In 2012, the non-regulatory federal agency within the U.S. Commerce Department’s Technology Administration NIST recommended a key length of 2048 for discrete logarithm group public key cryptography algorithms such as RSA in their document "Recommendation for Key Management" [9].

2.1.2 ECC

Elliptic Curve Cryptography was invented by Victor S. Miller and Neal Koblitz on independent works in 1985. First published by Miller in Use of Elliptic Curves in Cryptography [14] and two years later by Koblitz in Elliptic Curve Cryptosystems [15], it was proposed as "the analogue based on elliptic curves based on finite fields of public key cryptosystems which use the multiplicative group of a finite field" in Koblitz’s publication.

Both authors claimed cryptosystems based on elliptic curves to be more secure against known attacks to classical discrete logarithm based cryptosystems and faster that the latest. In both papers, algorithms analogue to the already then used protocols and cryptosystems are presented. In [14], an analogue to the Diffie-Hellman protocol described in the Diffie and Hellman publication [11] is proposed. In Koblitz work [15], analogues of the Massey-Omura and El-Gammal cryptosystems are described and exemplified providing a solution equivalent to these using Elliptic Curves.

ECC stands up as an alternative to RSA principally due to its higher level of security compared to RSA with the same key size. In Table 2.1 we show the equivalences in necessary key sizes for RSA and ECC to achieve the same level of security that a symmetric cryptography system would offer. ECC’s performance against RSA has been evaluated in Wireless Sensor Networks [8] resulting with ECC having an important advantage over RSA by reducing the computation time and the data amount transmitted and stored. In section 2.2, we describe Elliptic Curve Cryptography (ECC) in detail for the reader to get a deeper understanding of its mathematical foundations and its operations.
Table 2.1: Equivalences of necessary key sizes in bits for equivalent security level for different encryption schemes according to NIST guidelines [9] and ratio between ECC and RSA sizes.

<table>
<thead>
<tr>
<th>Bits of security</th>
<th>RSA (bits)</th>
<th>ECC (bits)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>160</td>
<td>1024</td>
<td>1:6</td>
</tr>
<tr>
<td>112</td>
<td>224</td>
<td>2048</td>
<td>1:9</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>3072</td>
<td>1:12</td>
</tr>
<tr>
<td>192</td>
<td>384</td>
<td>7680</td>
<td>1:20</td>
</tr>
<tr>
<td>256</td>
<td>512</td>
<td>15360</td>
<td>1:30</td>
</tr>
</tbody>
</table>

2.1.3 Diffie-Hellman key exchange protocol

The Diffie-Hellman protocol receives its name from Whitfield Diffie and Martin E. Hellman, who in 1976 proposed this protocol in their work [11]. The authors proposed the protocol as a method for public key distribution system that could use an insecure channel to perform the key distribution, as opposed to all the classical key distribution systems in which the keys need to be distributed using a previously secured channel. Their method used the discrete logarithm problem, on which public key cryptography relies, and used it as the base of the protocol. To establish a shared secret using a non-secured channel, the parties exchange their public keys and combine the other party’s public key with their own private key in an exponentiation manner and following the multiplicative groups fashion.

As described by Diffie and Hellman in their work: Let user A select a random number $X_A$ in the interval $0 < X_A < q$, keeping it secret and calculating

$$Y_A = \alpha^{X_A} \mod(q).$$

Let user B follow the same procedure, with $0 < X_B < q$.

$$Y_B = \alpha^{X_B} \mod(q).$$

When users A and B want to communicate, they use

$$K_{AB} = \alpha^{X_A X_B} \mod(q),$$

which they obtained by combining their secrets and the public information from the other party

$$Y_B^{X_A} \mod(q) = \alpha^{X_B X_A} \mod(q) = Y_A^{X_B} \mod(q) = K_{AB}$$

The information made public is $Y_A$ and $Y_B$. It is shown that, knowing one of the secrets, it is computationally easy to obtain the key $K_{AB}$ but, relying on the
discrete logarithm problem, it is not without the knowledge of $X_A$ or $X_B$. This allows to make public the necessary information and establish a secure channel using an insecure channel. One of the well known problems of the Diffie-Hellman protocol is that it doesn’t provide authentication. A man-in-the-middle attack can be performed, where an entity $C$ can intercept the communication and act as the legitimate party by providing a different public key to the legitimate parties and sitting as an intermediate point in their communications.

In the following section, we will focus on the Elliptic Curve Cryptography, its mathematical foundations and its operations.

2.2 Elliptic Curve Cryptography

As mentioned in the previous section, Elliptic Curve Cryptography (ECC) was invented in parallel by Victor S. Miller [14] and Neal Koblitz [15] in 1985. As proposed by the two authors, EC can be used to create a cryptosystem analogue to the Discrete Logarithm (DL) based systems.

The classical discrete logarithm problem is defined as follows in Elliptic Curves, Number Theory and Cryptography [16]:

Let $p$ be a prime and let $a, b$ be integers that are nonzero mod $p$. The classical discrete logarithm problem is to find an integer $k$ such as

$$a^k \equiv b \pmod{p}.$$ 

When switching to EC groups, EC cryptosystems are based on the Elliptic Curve Discrete Logarithm Problem (ECDLP). Defined by Koblitz as "Given an elliptic curve $E$ defined over GF($q$) and two points $P, Q \in E$, find an integer $x$ such that $Q = xP$ if such $x$ exists.", which is the EC equivalent to the classical discrete logarithm problem expressed in the common group additive notation used in elliptic curves.

In this section we will discuss first the mathematical foundations behind EC and ECC. After that, the operations of ECC will be defined and finally ECC will be compared with other crypto solutions and its choice will be reasoned.

2.2.1 Mathematical foundations

The most common form of expressing an Elliptic Curve is the Weierstrass Equation. An elliptic curve $E$ is the graph of an equation of the form

$$y^2 = x^3 + Ax + B$$ (2.1)

where $A$ and $B$ are constants. Usually, $A, B, x$ and $y$ are elements of a field such as the real numbers $\mathbb{R}$, the complex numbers $\mathbb{C}$, the rational numbers $\mathbb{Q}$, or a finite
field $K$. An elliptic curve is by definition non-singular. To satisfy this condition, the discriminant $\delta = -16(4A^3 + 27B^2)$ is required to be different than 0. This requires $A$ and $B$ satisfy the following:

$$4A^3 + 27B^2 \neq 0$$

The sign of the discriminant in the real representation of the curve determines the form of the graphical representation. Two examples can be seen in Figure 2.1.

The points with coordinates in some field $L \supseteq K$ are expressed by

$$E(L) = \{\infty\} \cup \{(x,y) \in L \times L \mid y^2 = x^3 + Ax + B\}.$$ 

The group law, or point addition is defined in [16] as follows: Let $E$ be an elliptic curve defined by $y^2 = x^3 + Ax + B$. Let $P_1 = (x_1,y_1)$ and $P_2 = (x_2,y_2)$ be points on $E$ with $P_1, P_2 \neq \infty$. Define $P_1 + P_2 = P_3 = (x_3,y_3)$ as follows:

1. If $x_1 \neq x_2$ then
   $$x_3 = m^2 - x_1 - x_2, \quad y_3 = m(x_1 - x_3) - y_1, \text{ where } m = \frac{y_2 - y_1}{x_2 - x_1}$$

2. If $x_1 = x_2$ but $y_1 \neq y_2$, then $P_1 + P_2 = \infty$.

3. If $P_1 = P_2$ and $y_1 \neq 0$, then
   $$x_3 = m^2 - 2x_1, \quad y_3 = m(x_1 - x_3) - y_1, \text{ where } m = \frac{3x_1^2 + A}{2y_1}$$

4. If $P_1 = P_2$ and $y_1 = 0$, then $P_1 + P_2 = \infty$.

Moreover, define

$$P + \infty = \infty$$
2.2. Elliptic Curve Cryptography

for all points \( P \) on \( E \). Also, the points on \( E \) form an additive abelian group with \( \infty \) as the identity element.

In Elliptic Curve Cryptography, the elliptic curves used are defined over a finite field \( F \) as it was already suggested by Miller [14]. Let \( F \) be a finite field and let \( E \) be an elliptic curve defined over \( F \). The group \( E(F) \) is the group of points belonging to the curve with coordinates \((x, y)\) belonging to \( F \).

The first parameter of the group \( E(F) \) that we can define is the order. Let \( \#E(F) \) be the order of \( E(F) \) defined as the number of points that form the group \( E(F) \). Also, let \( G \in E(F) \) be a point in the elliptic curve \( E \). The order of a point \( G \) is the smallest positive integer \( k \) such that \( kP = \infty \). Also, as a fundamental result of group theory (a corollary of Lagrange’s theorem [17]), it is known that the order of the point always divides the order of the group \( E(F) \).

With this, and according to RFC 9060 [10], in cryptographic contexts, an elliptic curve parameter set consists of a cyclic subgroup of an elliptic curve together with a generator point of the group. Working over a prime order finite field, with characteristic greater than three (other curves are out of the scope of the RFC and of this work), an elliptic curve group is completely specified by the following parameters:

- The prime number \( p \) that indicates the order of the field \( F_p \).
- The parameter \( A \) in the presented Weierstrass form of the curve equation.
- The parameter \( B \) in the presented Weierstrass form of the curve equation.
- The generator point \( G \) of the subgroup.
- The order \( n \) of the subgroup generated by \( G \).

2.2.2 Operations

2.2.2.1 Representation and coordinate systems

The operations over an elliptic curve in a cryptography context are associated to a particular parameter set. Operations using different points must be performed on points belonging to the same group.

The basic representation of the points is using the so called affine coordinates. It corresponds to the presented Weierstrass equations and in which the points are represented by two coordinates \((x, y)\). There are other coordinate systems that can be used for point representation in elliptic curves. Another commonly used type of coordinates are the projective coordinate systems. The projective coordinates allow the representation of the points in an elliptic curve in the projective space \( P_K \). These representations allow avoiding (multiplicative) inversions in the group
law, a costly operation in many platforms and a factor of high interest for this work. The two commonly used projective coordinate systems and implemented in this work are described in the IEEE Standard Specifications for Public-Key Cryptography [18] and are the following two.

- **In Homogeneous coordinates** an affine point \((x, y)\) is represented as \((X : Y : Z) = (θx, θy, θ)\) for some \(θ ≠ 0\). The neutral element (or point at infinity in EC notation) is given by \((0 : θ : 0)\). In homogeneous coordinates, all the three coordinates are said to have the same weight. The elliptic curve becomes, in this coordinate system, \(Y^2 = X^3 + AXZ^2 + BZ^3\).

- **In Jacobian coordinates** an affine point \((x, y)\) is represented as \((X : Y : Z) = (λ^2x, λ^3y, λ)\) for some \(λ ≠ 0\). In Jacobian coordinates, the \(X\) and \(Y\) coordinates are said to have weight 2 and 3 respectively. The elliptic curve becomes, in this coordinate system, \(Y^2 = X^3 + AXZ^4 + BZ^6\).

### 2.2.2.2 Point addition

The group operation or point addition has already been presented in section 2.2.1 for the affine coordinates. In the projective coordinate systems (homogeneous and jacobian) the modular inversions are exchanged by multiplications, increasing the performance in platforms on which the inverse operation (mainly division) is computationally costly. The use the alternative coordinate systems is suggested and analyzed in many references such as by Marc Joye in [19]. In Joye’s work a table comparing the number of operations required for point addition and point doubling (addition of the point to itself) in different coordinate systems is presented. We show the table partially regarding the coordinate systems implemented in this work in Table 2.2 where \(M\) represents the cost of a multiplication, \(S\) the cost of a squaring, \(I\) represents the cost of a modular inverse and \(c\) represents the multiplication by a constant.

<table>
<thead>
<tr>
<th>Coordinate system</th>
<th>Point addition</th>
<th>Point doubling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affine</td>
<td>(2M + S + I)</td>
<td>(2M + 2S + I)</td>
</tr>
<tr>
<td>Homogeneous</td>
<td>(12M + 2S)</td>
<td>(5M + 6S + 1c)</td>
</tr>
<tr>
<td>Jacobian</td>
<td>(10M + 4S)</td>
<td>(1M + 8S + 1c)</td>
</tr>
</tbody>
</table>

Table 2.2: Cost in operations of point addition and point doubling for different coordinate systems.

In many works such as [19] it is concluded that the use of the projective
coordinate systems increases the speed of the elliptic curve point addition in those systems with a computationally costly modular inversion operation.

2.2.2.3 Scalar multiplication

The scalar multiplication operation is the basic operation of the Elliptic Curve Cryptography. It is the equivalent of the exponentiation in the additive notation and it consists of performing point additions repetitively. Due to the duality with the exponentiation, one of the first algorithms that arises for the implementation of this operation is the binary exponentiation algorithm. The duality can be established by, instead of the multiplication operations, an addition is performed, obtaining at the completion of the algorithm the result of the scalar multiplication.

2.2.2.4 Key generation

A key pair in the ECC context consists, as in all the other public key schemes, of a private and a public key. The private key consists of an integer randomly selected and that is used to create the corresponding public key of the key pair. To generate the public key, the scalar multiplication is performed. The selected secret key is the scalar and the point is the selected generator point of the selected EC group.

Let \( G \) be the generator of the elliptic curve group selected, \( p \) the prime number indicating the order of the field, and \( K_{A_{\text{priv}}}, K_{A_{\text{pub}}} \) the private and the public key of an entity A respectively. The key pair is generated as follows:

1. **Private key** \( K_{A_{\text{priv}}} \). A random number \( x \) is selected such as \( 0 < x < p \).
   \[ K_{A_{\text{priv}}} = x. \]

2. **Public key** \( K_{A_{\text{pub}}} \). A scalar multiplication is performed using the private key and the generator point of the group.
   \[ K_{A_{\text{pub}}} = K_{A_{\text{priv}}} * G. \]

2.2.2.5 Elliptic Curve Diffie Hellman (ECDH)

The Elliptic Curve Diffie Hellman (ECDH) protocol was presented by Victor S. Miller in [14]. There, Miller states that the Diffie-Hellman protocol “really only uses the property that we are working in a group [...]” and that “the points on an elliptic curve have the structure of an abelian group. Thus we may make the analogous constructions over elliptic curves”. In the ECDH protocol, this steps are followed:

1. The parties involved, A and B have previously agreed on a EC parameter set.

2. The parties exchange their public keys, consisting of a point in the EC.
3. Each party combines the received public key with its private key in a scalar multiplication operation. The result of that operation is another point in the EC. That result is the shared secret resulted as an output of the protocol. It is common practice to take the x coordinate of the point and derivate a session key from that value in a manner agreed by both parties participating in the protocol.

2.2.2.6 Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve version of the Digital Signature Algorithm (DSA) is a variant on the ElGamal signature scheme [20]. The algorithm allows the use of ECC to generate and verify signatures for messages. Let \( r \) be the order of the base point \( G \) in the selected EC set. Let the receiver know the signer public key \( K_{A_{publ}} \).

Basically the algorithm used is the following [16]:

**Signature generation for message \( m \).**

1. Choose a random integer \( k \) with \( 1 \leq k < r \) and calculate \( R = kG = (x, y) \).
2. Calculate \( s = k^{-1}(m + K_{A_{priv}}x) \pmod{r} \).

The complete signed document sent to the receiver is \( (m, R, s) \).

**Signature verification for \( (m, R, s) \).**

1. Calculate \( u_1 = s^{-1}m \pmod{r} \) and \( u_2 = s^{-1}x \pmod{r} \).
2. Calculate \( V = u_1G + u_2K_{A_{publ}} \).
3. Declare the signature valid if \( V = R \).

If the verification is successful, the verification equation holds:

\[
V = u_1G + u_2K_{A_{publ}} = s^{-1}mG + s^{-1}xK_{A_{publ}} = s^{-1}(mG + xK_{A_{priv}}G) = kG = R.
\]

One of the main differences between ECDSA and the ElGamal system is that the ECDSA only uses two scalar multiplication operations for the signature verification operation, whereas ElGamal uses three. Since it is the costly operation of the algorithm, ECDSA shows an improved efficiency over ElGamal.

2.2.3 Comparison/Advantages

ECC offers a big advantage when compared to RSA. It requires a much smaller key size to achieve the same level of security. As shown in the Guide to Elliptic Curve Cryptography by Hankerson et al. [21] and shown previously in Table 2.1,
to achieve the security level of a 128-bits symmetric encryption, RSA requires
a key length of 3072 bits, whereas ECC achieves the same security by using
lengths of only 256 bits. This is a big advantage of ECC over RSA and as
the minimum security requirements increase, the ratio of the required key sizes
between RSA and ECC will keep increasing, converting ECC into an even better
alternative to RSA or until the point when RSA key sizes are not able to be handled
anymore. This fact also makes ECC a good candidate for systems with memory
requirements such as the ones used in Wireless Sensor Networks (WSN) since the
smaller size of the managed variables makes ECC more suitable for this kind of
systems.

It was previously mentioned that ECC performs operations on variables of
the size of the key. This size is not usually less than 160 bits. In the current
architectures and programming languages, arithmetic operations are performed
in on variables of a size corresponding to the processor architecture, that is 32
bits or 64 bits in the more recent architectures. This fact creates the need of
an arithmetic able to handle variables of such sizes and manipulate and perform
arithmetic operations on them. For this reason, the need for an arbitrary precision
arithmetic arises.

2.3 Arbitrary precision arithmetic

Arbitrary precision arithmetic refers to operations or arithmetic performed on
numbers that have arbitrary high precision. The basic operations in an arithmetic
are addition, subtraction, multiplication and division. To build a multiple or
arbitrary precision arithmetic, primitive operations on one-place integers are used.
In The Art of Computer Programming by Knuth [22], we can find a basic reference
containing the algorithms for the construction of this so called arbitrary precision
arithmetic. The following primitives are given to work with:

- Addition or subtraction of single-precision integers, giving a single-precision
  result and a single precision carry.

- Multiplication of a single-precision integer by another single-precision
  integer, giving a double-precision integer.

- Division of a double-precision integer by another single-precision integer,
  provided that the quotient is a single-precision integer, and yielding also a
  single-precision reminder.

Using the mentioned primitives, the operations for an arbitrary precision arithmetic
can be built, achieving the following operations:
• Addition or subtraction of n-precision integers, giving a n-precision answer and a single precision carry.

• Multiplication of an m-precision integer by an n-precision integer, giving an \((m+n)\)-precision result.

• Division of an \((m+n)\)-precision integer by an n-precision integer, giving an \((m+1)\)-precision quotient and an n-precision remainder.

2.3.1 Basic operations

2.3.1.1 Multiplication and squaring

Multiplication is an operation performed very commonly in ECC operations such as point addition. Therefore, an efficient algorithm to perform it is important. Efficient algorithms for multiple-precision multiplication trying to reduce the number of single-precision multiplication used. In the HoAC [12] it is proposed an algorithm to perform a multiplication of an m-precision integer by an n-precision integer that requires \((m+1)(n+1)\) single-precision multiplications.

Squaring is a special multiplication case but very commonly used in ECC operations. Therefore, a special effort to improve its efficiency can result in an overall improvement of the performance. In [23] an efficient squaring method is proposed. It is based on the algorithm proposed in the HoAC and relies on the fact that a squaring can be seen as a symmetrical multiplication where the inner products are repeated and therefore can be reused. As a result, the operation results in approximately one half of the single-precision operations that would be performed using the general multiplication algorithm.

2.3.1.2 Division

Due to the lack of an efficient division operation in the current processors apart from single-precision division, there has been an important amount of work to improve the performance of this operation. In Improved division by invariant integers [24], a method using a pre-computed approximation of the reciprocal of the divisor is proposed to exchange the division by a multiplication and a few adjustment steps. The method proposed by Möller and Grandlund consists in the calculation of an approximate of the reciprocal of the divisor to replace the division operation by a multiplication and finally a limited number of steps are performed to adjust to the correct result. In they work they show that their proposed method shows a significant improvement compared to earlier methods in operations on multiple-precision variables. This comes of an extreme value for the operations performed in ECC due to the modular nature of the operations,
since division is an operation that will clearly define the performance of the overall operations.

2.3.1.3 Special operations

To be able to perform all the necessary operations for the ECC the arithmetic needs to provide some special operations such as the Greatest Common Divisor operation and the Extended Euclid’s algorithm. These operations are required in the modular arithmetic operations and, more precisely, in the modular inversion operation. Efficient algorithms for this operations are presented in [22] and have been widely known and used in other public-key cryptography schemes such as RSA.

2.4 Contiki

The Contiki OS is an open source operating system designed for the Internet of Things and that has been developed in SICS and all around the world since 2003 [4]. Contiki is a light-weight OS specially designed to use a very small amount of resources and it provides, in its basic distribution, a complete IP stack also developed at SICS called uIP. One of Contiki’s main characteristics is the use of protothreads.

2.4.1 Protothreads

A remarkable characteristic of the Contiki OS is that it has only one stack for all the processes. This is possible thank to the use of protothreads to achieve a thread-like behavior without a real multi-thread infrastructure and overhead. Protothreads are a programming abstraction used in Contiki that allows to write in a thread-like style. This is achieved by the use of macros that the C language provides in its precompiler directives or the switch statement. The Contiki OS code is all written in C language and protothreads provide conditional blocking wait statements using only C language constructs, which allows to avoid the use of state machines with the memory overhead that they add.

All the processes in Contiki start and end respectively with the macros PROCESS_BEGIN and PROCESS_END. A process contains a process thread that is a single Contiki protothread. The macros used to implement them are the core of Contiki and in Code 2.1 and 2.2 are shown the C preprocessor implementation of the protothread main operation and the local continuations implemented using the C switch statement as presented in the Contiki wiki. Protothreads and their insights are extensively described in [25].
struct pt { lc_t lc };
define PT_WAITING 0
define PT_EXITED 1
define PT_ENDED 2
define PT_INIT(pt) LC_INIT(pt->lc)
define PT_BEGIN(pt) LC_RESUME(pt->lc)
define PT_END(pt) LC_END(pt->lc);
return PT_ENDED
#define PT_WAIT_UNTIL(pt, c) LC_SET(pt->lc); \ if(!(c)) \ return PT_WAITING
#define PT_EXIT(pt) return PT_EXITED

Code 2.1: C preprocessor implementation of Contiki OS protothread macros and implementation of local continuations.

typedef unsigned short lc_t;
define LC_INIT(c) c = 0
define LC_RESUME(c) switch(c) { case 0:
define LC_SET(c) c = __LINE__; case __LINE__: 
define LC_END(c) }

Code 2.2: C implementation of local continuations using switch statement.

2.4.2 Code, compile and upload Contiki

All Contiki processes follow the same structure. As mentioned in the previous section, all processes have a process thread, which is a Contiki protothread called from the poller and that contains the process code. With the described macros, the thread-behavior is achieved. Following there is a list of the protothread macros that are used in the Contiki code as explained in the Contiki Wiki [26].

- PROCESS_BEGIN() : Start of the protothread.
- PROCESS_END() : End of the protothread.
- PROCESS_EXIT() : Exit the process.
- PROCESS_WAIT_EVENT() : Wait for an event of any kind.
- PROCESS_WAIT_EVENT_UNTIL() : Wait for an event and the condition to be true.
- PROCESS_YIELD() : Wait for an event of any kind.
2.4. CONTIKI

- PROCESS_WAIT_UNTIL() : Wait for the condition to be true; may not yield the process.
- PROCESS_PAUSE() : Temporarily yield the process.

The use of these macros allows the event-driven/threaded behavior that characterizes Contiki and makes it capable to achieve the benefits of an event-driven system without maintaining complex state machines.

```c
/**
 * A very simple Contiki application showing how Contiki programs look
 * \author Adam Dunkels <adam@sics.se>
 */
#include "contiki.h"
#include <stdio.h> /* For printf() */

PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);

PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();

    printf("Hello, world\n");

    PROCESS_END();
}

Code 2.3: Example code of the Hello World application included in Contiki 2.7.

Contiki supports a variety of different platforms such as TMote Sky, STM32w SoC, Zolertia z1 motes and Texas Instruments MSP430x among others. Each Contiki application has its own Makefile indicating the location of the Contiki installation, and includes the system Makefile it allows the use of the apps in the Contiki apps folder easily by simply adding the necessary apps to the variable in the Makefile.

For compilation, the make command can get as an argument the TARGET platform, that is defined to compile the code for the selected system. For example,
to compile the Hello World example application for the Z1 mote platform, the following command can be used in the examples/hello-world folder: `make TARGET=z1 hello-world`. If the device is actually connected to the compiling machine the code can be compiled and directly uploaded to the device to start running it adding the suffix `.upload` to the project name. Following the previous example the command used would be: `make TARGET=z1 hello-world.upload`

2.5 Related work

There are other open source operating systems designed for low-power operated devices. One of the most well-known is TinyOS. TinyOS [27] is a lightweight Operating System using an event-driven model and it is written in nesC, a dialect of C. There also exists already an Elliptic Curve Cryptography library for low-power devices for TinyOS, TinyECC. TinyECC [28] was presented in 2008 by An Liu and Peng Ning and it provides many optimizations to the classical ECC operations to improve the efficiency and adapt them for the use in low-power devices and the TinyOS system while providing a very configurable ECC solution.
Chapter 3

Design and implementation

In this chapter the design and implementation of this work will be discussed. The work is divided into two main parts. First, the design and implementation of an arbitrary precision big integer arithmetic library (Bigint for short) in Contiki OS to allow calculations using arbitrarily big numbers, which will be the base for the Elliptic Curve Cryptography library. After that, the ECC library itself will be discussed, with its main operations that provide a public key cryptography solution based on Elliptic Curves.

3.1 Design

In this work, the design started on the arithmetic for unsigned integers library, which is a vital part of the implementation and the element which the ECC library and its operations lay on top of. The bigint library is designed in this work to provide all the basic operations for an arithmetic supporting an arbitrary precision, which is defined at compile time, that fits the needs of the application that is going to make use of it. This design was conceived to allow the programmers to select the size of the manipulated variables, both the precision of the arithmetic and the size of the basic units or words that the processor manages individually to fit different processor architectures and their needs.

As explained in the background chapter 2, the Contiki OS is an operating system especially designed for systems with relevant restrictions and one of them is usually the available memory. For this reason the solution is designed to use the minimum amount of memory possible, always prioritizing this parameter against speed or performance. In the literature study performed to establish the design and select the algorithms for the implementation, this was always the main parameter taken into account and lead all the research towards the design and implementation of this work.
The design of the ECC library followed a similar approach but, in this case, the literature study showed that the ECC operations were clearly defined and that there was not much possibility of manipulation and modification of the main procedures. What was shown in the analysis was that the coordinate system used to represent the points in the elliptic curves and operate on them could have a big impact in the performance and the memory of the implementation. To be able to provide a good comparison and evaluation, the ECC library was designed to allow us the use of the different coordinate systems and to perform an evaluation of the performance of them and, in the future, being able to add different coordinate systems that could improve the implementation or offer other advantages that could satisfy other requirements or provide different performance in certain aspects that the programmer could be interested in.

The design of this work was done having in mind the flexibility of the libraries. As explained previously, different word sizes are allowed in the bigint library to adapt the operations to the platform and architecture used. The ECC library allows on its turn the use of different key sizes and curves. The curve parameters are simply included as a file for the application and the ECC library implementation is totally independent of the curve used. In figure 3.1 we can see the basic files of the libraries and the relationship between them.

![Figure 3.1: C files contained in the libraries and usage relationship between them.](image)

### 3.2 Implementation

#### 3.2.1 BigInteger library implementation

The first part of the work of this thesis was to implement an arithmetic module that was able to handle unsigned integers of arbitrarily precision, or the also so called big integers. The semantics provided by C, and therefore in the Contiki OS allow to perform arithmetical operations on the C variables, which are limited by the processor used and the C language itself. As discussed earlier in chapter 2, the size of the keys used in ECC is at least 160 bits and all the operations are performed on numbers of the order of the key size. Therefore, the need to be able
to execute arithmetic operations on the big integers arises. The implementation of the bignum library in this work was divided into two main parts, first the basic operations of an arithmetic were implemented. After that, support for modular operations was added, using these aforementioned operations as ground operations. Finally, some other manipulation algorithms were implemented to provide a the support needed by the ECC library.

### 3.2.1.1 Basic operations

This library implements the basic arithmetic operations for unsigned integers of arbitrary precision, that is, the size of the variables representing a number is only limited by the available memory in the system. The basic units in the implementation are called u_word, referring to the unsigned word that corresponds to a variable defined at compile time. This implementation supports 2 sizes of u_word, 16 bits and 32 bits, corresponding to the Standard C variables uint16_t and uint32_t introduced in the C99 standard. The library performs the arithmetic operation on arrays of this basic units taking as input a pointer to the array of the aforementioned basic units u_word and the digits, that is the number of basic units that the array consists of. For the implementation of the different operations, and after the literature study to select the most suitable algorithms for the purpose of this work, this thesis implements the following set of operations to be able to manipulate and provide the operations required for the ECC library to perform the different cryptography procedures.

Following, we present a list of the most relevant basic operations and provide a brief explanation of the implementation and the used algorithms. Further references to the algorithms can be found in Chapter 2.

- **Addition**: Implemented following the schoolbook addition algorithm, adding each word and keeping track of the carry. Returns the result in a variable of the same size and the carry.

- **Subtraction**: Implemented following the 2’s binary complement method. This consists on calculating the 2’s complement of the subtrahend and calculating the binary addition of the complement and the minuend. The operation returns a carry if the result is negative.

- **Squaring**: Implemented using the algorithm presented in [23] that uses the "symmetries" in the multiplication operation due to the fact of both operands being the same number.

- **Multiplication**: To implement the multiplication operation the schoolbook multiplication algorithm was followed as presented in HoAC [12]. It allows
the product of two numbers of different size and uses the single-precision product as the basic operation with a variable of double precision.

- Division: The division is one of the core operations of the library and becomes a basic operation for the modular arithmetic. The implementation is based on the algorithms presented in [24]. The division is calculated using the reciprocal method, which consists on calculating the reciprocal of the divisor and using this value to calculate an approximate result of the division and adjusting it to find the correct result. The algorithm solves the division using multiplications which are less costly operations and achieves the correct adjustment with a limited number of iterations. The implementation of the computation of the inverse and the division of a double-precision by a single-precision variables can be seen in algorithms 3.1 and 3.2 respectively. In the latter, the calculated reciprocal is used to calculate an approximation of the result by multiplying and then adjust it if the result is not the correct.

```
#define MAX_BIGINT_WORD 0x7FFFFFFF

u_word
reciprocal(u_word * d)
{
    u_doubleword aux = MAX_BIGINT_WORD - (*d);
    aux = aux << BIGINT_WORD_BITS;
    aux += MAX_BIGINT_WORD;
    return (u_word) (aux / (*d));
}
```

Code 3.1: Implementation of the calculation of the division reciprocal.

```
u_word
basic_division(u_word * u, u_word * d, u_word * q,
              u_word * v)
{
    u_word q_aux[2];
    u_word r;

    bigint_basic_mult(q_aux, *v, u[1]);
    bigint_add(q_aux, q_aux, u, 2);
    q_aux[1]++;
    r = (u_word) (q_aux[1] * (*d));
    r = u[0] - r;
    if(r > q_aux[0]) {
```

Code 3.2: Implementation of the basic division.
3.2. IMPLEMENTATION

Code 3.2: Implementation of the double-precision word by single-precision word using division reciprocal

```c
q_aux[1]--;
r = r + (*d);
}
if(r >= (*d)) {
    q_aux[1]++;
r = r - (*d);
}

if(q != NULL)
    q[0] = q_aux[1];
return r;
```

3.2.1.2 Modular arithmetic

The basic operation in modular arithmetic is the remainder calculation or modulo. As previously defined, being \( a \) and \( n \) two positive integers, \( a \) modulo \( n \) (or \( a \mod n \)) is the remainder of the euclidean division of \( a \) by \( n \). The modulo operation can be purely implemented by performing a division and returning as a result the remainder of the latter. Using this, a very simple modular arithmetic can easily be built on top of the operations of the arbitrary precision arithmetic. In this work, the simple and plain modulo operation is implemented using the division but to improve the performance of the library and avoid costly division operations, other algorithms have been used.

- Remainder calculation (modulo): Implemented as a division and returning the remainder of the operation.

To avoid the use of the division operation, which we know is computationally costly in this architecture, we implemented the modular operations by trying to avoid division and exchanging them for additions and subtractions whenever it was possible. The following precondition is required for the operations. If we define \( a \) as the result, \( b \) (and \( c \) potentially) as the two operands and \( n \) as the modulo, it is required that \( b = b \mod n \) and \( c = c \mod n \).

- Modular addition: Due to the mentioned precondition, we can ensure that \( b + c < 2n \). This allows to perform the modulo operation after the normal addition by just subtracting \( n \) from the result if the addition returned a carry or the result is bigger than \( n \), exchanging a division to obtain the modulo by, in the worst case, a single subtraction.
• Modular subtraction: Using the properties of the modulo we can ensure that \((b - c) \mod n = n - (c - b) \mod n\). This allows to perform the subtraction as follows: If \(b >= c\) a normal subtraction can be performed, \(a < n\) is guaranteed by construction. If \(c > b\), the aforementioned property is used and only 2 subtractions need to be performed to ensure the modular operation.

• Modular multiplication: This operation is performed using the normal multiplication and applying the modulo operation after.

• Modular power: This operation is performed using the normal power and applying the modulo operation after.

• Modular inverse: Implemented using the algorithm from Art of Computer Programming [22] and using the Extended Euclid’s algorithm implementation.

3.2.1.3 Manipulation algorithms

To complete the arithmetic library some manipulation algorithms that are needed also for the ECC library were implemented. The greatest common divisor, a very well known and widely used algorithm was implemented following the algorithm proposed in Knuth [22]. We also implemented and used in the ECC library a binary variant of the algorithm also proposed in Knuth. In the latter, simpler operations are used, increasing the speed of the iterations. Related to this, the extended Euclid’s algorithm was implemented. Using the algorithm proposed in the same work but adding a modification for the calculations since the modular inverse operation doesn’t need the complete result since the modulo used in the algorithm is always the same.

3.2.2 Elliptic Curve Cryptography library implementation

For the implementation of the Elliptic Curve Cryptography (ECC) library, the basic reference was the RFC 6090 [10]. In this RFC we find the fundamental algorithms of ECC defined from 1994 and before that allow the implementation of the basic algorithms. The scope of the RFC and of this work is on curves defined over fields of characteristic greater than 3.

The basic operation on EC is the addition of points. It will be the ground operation for the scalar multiplication which, at its turn, becomes the core operation on the ECC operations: Key generation, Elliptic Curve Diffie-Hellman key exchange (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA).

In the aforementioned RFC two different coordinate systems are proposed: affine coordinates and homogeneous coordinates. In this work we also add the
3.2. IMPLEMENTATION

The algorithms for the ECC operations are clearly specified. As previously stated, in our work we try to minimize the memory usage so the implementations aims minimize the number and size of variables used to perform the different operations. The three mentioned coordinate systems are implemented to allow comparison in the memory usage and performance of the operations.

3.2.2.1 EC point addition

Point addition is the group operation associated with the elliptic curve group. As previously explained in chapter 2.2, in this work we use the additive notation. The group operation was explained in section 2.2.2.2 and will define the core of the EC operations. The implementation is done in the 3 different coordinate systems and they are completely different and determinant. As mentioned, the operations are designed to use the minimum number of variables and size and, therefore, the proposed algorithms are twisted to reuse variables as much as possible and avoiding therefore unnecessary usage of memory. In this work, affine coordinates are always the basic and default representation of the points in an elliptic curve.

- Affine coordinates. The addition operation using these coordinates in this work is the basic implementation directly derived from the elliptic curve fundamentals and as defined by Bender and Castagnolli in [29].

- Homogeneous coordinates. Following the mentioned RFC 6090, the implementation of the homogeneous coordinates is done in this work as defined by Koyama et al. [30]. Again in this case, the implementation is designed to use the minimum number of variables, reusing the allocated memory as much as possible.

- Jacobian coordinates. For this coordinate system, the algorithms proposed in [18] have been used. In this case, we distinguish between the point doubling and point addition (of different points). The number of operations reduces considerably if it is a point doubling and this is a very common operation in the scalar multiplication method used in this work so it is important to optimize as much as possible this operation. Once again, the proposed algorithm has been twisted to reuse the variables, even though in this case the algorithm has already been strongly optimized for that matter.
3.2.2.2 Scalar multiplication

The scalar multiplication algorithm used in this work follows the principles of the widely known and used binary exponentiation algorithm. Thanks to the duality addition/multiplication notation for EC operations, exponentiation algorithms can be used to perform our scalar multiplication operation. In this work, the repeated square-and-multiply algorithms are used.

For the affine and homogeneous coordinate system, a right-to-left binary exponentiation algorithm is followed. We follow the idea of the algorithm proposed in [12] to compute \( R = kG \) where \( R \) is the point result of the operation, \( k \) is the scalar and \( G \) is the point of the EC, but applying our additive notation instead of the multiplicative notation of the exponentiation. The considered algorithm is used, following the addition notation and scanning across the scalar (the exponent in the multiplicative notation) instead of dividing by two (or shifting it in binary notation) to avoid the shifting operation and avoid the creation of a new variable to hold the modified scalar.

The scalar multiplication is the same operation for the three coordinate systems. To allow that in our implementation, before using the EC addition, the coordinates are transformed to the corresponding coordinate system and converted back to the affine system at the end of the scalar multiplication operation. That allows using the advantage that the projective systems offer in the addition operation and perform the conversion only at the beginning and the end of the scalar multiplication operation.

For the jacobian scalar multiplication, we also implemented a variation of the presented operation: the left-to-right binary exponentiation. In this case, one of the additions has always as an operand \( G \) and, for some choices of \( G \), the addition operation can be computed more efficiently and therefore improving the performance.

In the following subsections the ECC operations that this library implements can be found. They use the EC operations that we just exposed as the ground blocks to perform the EC based cryptography.

3.2.2.3 Key pair generation

The key pair generation operation consists on obtaining a key pair consisting of two different keys: the private and the public key. The private key is a random uniformly distributed integer between 1 and the order of the selected elliptic curve. In our implementation we use the Contiki OS random generator that, in the STM32W platform, uses the radio as a source to obtain random values.
3.2. IMPLEMENTATION

3.2.2.4 EC Diffie-Hellman exchange

As discussed in 2, the analogue to the Diffie-Hellman algorithm for ECC was proposed by Miller in [14]. The Elliptic Curve Diffie-Hellman protocol (ECDH) uses the same principle as the classic DH. In our implementation, providing an elliptic curve point and a scalar (the public key of B and the private key of A) to the secret generation function, A can derive the shared secret to communicate with B that comes as the result of the Diffie-Hellman protocol.

First the provided point (B’s public key) is verified to be belonging to the agreed EC between A and B. If that condition is satisfied, the point is combined with the private key of A on A’s side and vice versa (The point obtained from A and the private key of B) on B’s side. This combination is achieved using the presented scalar multiplication operation, using the private key of the party as the scalar and the public key of the other party as the point to be multiplied.

3.2.2.5 ECDSA: Elliptic Curve Digital Signature Algorithm

The implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA) in this work is based in the algorithm proposed in [16]. The algorithms for signature generation and signature verification can be seen in Algorithm 1 and Algorithm 2 respectively. As it can be seen in the algorithm, the implementation of the signature generation requires less computational effort than the verification. Especially, the generation operation only requires one scalar multiplication, whereas the verification requires two scalar multiplications.

The implemented algorithms correspond to the Type I KT signature defined in the RFC 6090 and it is the EC equivalent to the Digital Signature Algorithm (DSA). In this case we have two different operations, the signature generation and the signature verification.

In ECDSA, a message digest or hash of the message is used to create the signature. In RFC 6090 it is stated that "any collision-resistant hash function is suitable for use in KT signatures". Also the RFC states that the number of bits in the output of the hash should be equal or close to the number of bits needed to represent the group order. In the same document some hashes are recognized as suitable and one of them is SHA-256, which is used in this implementation.

In the signature operation, the order \( r \) of the generator of the group \( G \) is used as the modulo for the different operations. This implementation only supports curves that use the same number of words or less than the order of the group to be represented, which is the case in most of the commonly used elliptic curves for
cryptography purposes.

**Algorithm 1:** Implemented EC signature generation algorithm. It consists of a modular inverse operation, an addition and one scalar multiplication.

**Input:** Message to be signed $m$, EC parameters (generator point $G$, order $r$), secret key $a$  
**Output:** $(m, R, s)$ forming the signed message

1. Select a random integer $k$ with $1 \leq k \leq r$
2. Compute $R = kG = (x, y)$
3. Compute $s = k^{-1}(m + ax)(mod r)$

**Algorithm 2:** Implemented EC signature verification algorithm. It consists of a modular inverse operation, multiplications and two scalar multiplications.

**Input:** Signed message to be verified $(m, R, s)$, EC parameters (generator point $G$, order $r$), sender public key $Q$
**Output:** Validity/invalidity of the signature for the message $m$

1. Compute $u_1 = s^{-1}m(mod r)$ and $u_2 = s^{-1}x(mod r)$.
2. Compute $V = u_1G + u_2Q$
3. if $V = R$ then
4.  Signature is valid
5. else
6.  Signature is invalid
Chapter 4

Evaluation

In this chapter the performance of the work produced in this thesis will be evaluated. Firstly, the evaluation set-up we used will be described. After that, the parameters used to analyze the performance of each of the operations will be defined and the different possible configurations listed. Finally we present and compare the results of the evaluation.

4.1 Evaluation set-up

All the tests have been performed on the STM32W108CC IEEE 802.15.4 wireless system-on-chip [31]. Some of the main characteristics of this chip are:

- 32-bit ARM® Cortex™-M3 processor
- 256-Kbyte Flash, 16-Kbyte RAM memory
- Operation at 6, 12 or 24 MHz.

The stand-alone tests have been performed using the STM32WC-RFCKIT, a board provided with the STM32F103 micro-controller at a frequency of 24 MHz that allows the connection via USB for programming and debugging purposes.

For the tests where 2 different entities exchange information, nodes from Yanzi Networks AB were used. These nodes also use the STM32W108CC chip and ContikiOS.

4.2 Evaluation parameters and configurations

Three main parameters are analyzed in this thesis for the evaluation of the solution:
• Computation time: To obtain the computational time of each operation, the Contiki OS clock library was used, taking the number of clock ticks provided by the OS at the beginning of each operation, comparing it to the count at the end of every operation and transforming that result into time units.

• RAM memory usage: We analyze the RAM usage by dividing it into different categories. The first one is the static allocated memory, which is obtained at compile time and consists of the static defined variables used. The second category is the stack memory. To determine the stack usage in every operation, a known pattern was written in the corresponding RAM area at the initialization of the Contiki OS. After the execution of the operation to be evaluated, the memory was scanned to determine the size of the zone that remained unmodified and therefore obtain the maximum usage of RAM.

• Flash memory usage: Obtained at compile time. To obtain it, we used the utility arm-gcc-eabi-size after the compilation of the testing application.

These three parameters give an accurate insight to the performance of the operations and the requirements that the library sets for the systems that will make use of it. Also these are 3 very important parameters in the wireless sensor networks paradigm where, as previously discussed, resources are usually very limited and valuable.

The performance of the ECC operations was evaluated using the 3 different coordinate systems that this thesis implements: affine, homogeneous and jacobian. As explained in chapter 3 the use of different coordinate systems allows to perform ECC operations in a totally different way, such as exchanging divisions and inversions for multiplications, which as it will be shown, has a crucial influence in the performance of the system.

Finally, an important parameter that defines the level of security and, of course, the performance of the system, is the key size. As it is shown in the presented results, the size of the key affects directly to the performance of the system in terms of memory and speed, establishing a trade-off between the desired level of security and the desired performance. Results regarding different key sizes are also presented and compared.

4.3 ECC operations evaluation

In this section, the evaluation of the ECC operations is discussed. As explained in the previous chapter, the main operation in ECC is the scalar multiplication.
It is the ground operation for all the different ECC functions as it is the main block in all of them, and also the most costly computation time-wise. Therefore, it will be determining in the evaluation results that will be presented in this section. In this implementation, the scalar multiplication operation is always performed following a binary exponentiation algorithm as explained in section 2.2.2, which consists on multiple additions of 2 points, and makes the addition the operation that is continuously and intensely performed. As it is shown in this section, there is a big difference in the performance of this operation depending on the coordinate system used as this addition is performed in very distinctive manners in the different coordinate systems, resulting in a considerable increase of performance by the use of the jacobian coordinate system. Since the computational complexity depends directly on the number of bits, the key size also determines the performance of the system.

In this work, the performance of the operation has been evaluated on the previously described set-up and the results that we obtained are represented by the mean values and the standard deviation with data obtained from 20 runs of the experiment in each of the 3 different coordinate systems.

4.3.1 Key pair generation

The key pair generation is an operation that every node performs independently and after the establishment of the elliptic curve parameters to be used (as described in section 2.2.2) this operation requires no input data from any other entity. Therefore, we evaluate this operation on a single node basis regardless of the presence of any other nodes in the network or whom it might potentially communicate with. The evaluation of the key pair generation has been performed on the described stand-alone set-up consisting of the STM32W-RFCKIT. The results of the evaluation of the key-pair generation can be seen in Figure 4.1.
From the results can be inferred that the selection of the coordinate system is a crucial factor for the performance of the system. Observing this results it becomes clear the expected effect of the computational cost of the division operation in this architecture and how the use of a projective coordinate system that substitutes divisions for multiplications can boost the performance of the operation in a factor of 4 when comparing the 2 extreme cases (affine and jacobian).

As the nature of the used algorithm suggested, 2.2.2, we can also observe a clear relationship between the key size and the time and memory consumed by the operation. The exponential scalar multiplication algorithm traverses all the bits in the multiplication factor and realizes an operation for every bit, creating a direct relationship between the key size and the number of operations performed.

It can also be seen that, since all the points of the curve and therefore the variables used to operate with them have the same size as the key, this produces the same direct relationship effect in the RAM memory usage as it will be seen in section 4.3.4.
4.3.2 EC Diffie-Hellmann exchange

As previously explained in 2.2.2, the Diffie-Hellman exchange involves two different entities in the process, as they exchange their public keys to obtain a shared secret to be used in a symmetrically encrypted communication. Therefore, one of the set-ups for the evaluation of this section includes nodes that communicate with a gateway to establish a secure communication channel.

For the first part of the evaluation, the Yanzi Networks AB [32] nodes and gateway were used to obtain this performance results. In this set-up, the node and the gateway exchanged their public keys, generate the shared secret corresponding to their private keys, and finally use that value to create a symmetric key for the secure communication channel.

In the Yanzi implementation, the key size used is 256 bits, using the curve secp256r1 defined in the SEC 2: Recommended Elliptic Curve Domain Parameters document by the Standards for Efficient Cryptography Group [33]. The symmetric cryptography algorithm used is AES-128 [34], using as key a secret derived from the result of the Diffie-Hellman exchange. In this real application set-up, it has been evaluated the establishment of the shared secret and therefore a secure channel for communication between a Yanzi gateway running in a Java environment and the Contiki based node with the STM chip. The establishment takes place between a node that is a network neighbor of the gateway and also in a multihop network where the exchange is performed between the node and the gateway with a second node in between, adding an additional hop in the network.

First, we present the public key exchange latency in 2 different scenarios. The first set-up consists of a single-hop network were the gateway and the node are neighbors and the messages are not routed through another node. In the second scenario, an intermediate node is added ensuring that the destination node and the gateway don’t have single hop route and all the messages must pass through the routing node in between becoming a two-hop network. In both scenarios, the latencies are measured in the Yanzi gateway since it is the entity starting the exchange and provides its public key and requests the other party’s corresponding key.
In this scenario, the solution provides an application layer secure channel and the network layer remains totally agnostic of the encryption layer. As it can be seen in the previous results, that results in an increment of latency due only to the additional hop in the network and the more potential re-transmissions of packets, but the intermediate node doesn’t process any data related with the encryption as it is in a layer that remains above the network processing. This makes the ECDH protocol independent of the network configuration allowing its use in deep networks as all the intermediate nodes act simply as relays of the protocol messages.

Finally to evaluate the impact of the different key sizes (since the Yanzi solution supports only 256 bits ECC) and the memory usage of the stand-alone application, we perform the tests using the stand-alone ECC-test application that only generates the shared secret corresponding to the provided keys.
4.3. ECC OPERATIONS EVALUATION

Figure 4.3: Computational time for EC Diffie-Hellman secret generation for different key sizes and using the 3 different implemented coordinate systems in the Contiki OS node.

In this results we can observe the same pattern as the previous section. There exists a clear dependency on the key size due to the nature of the scalar multiplication operation that lies in the heart of the ECC operations. Also again the results show the big impact that the coordinate system selection has on the overall performance of the operation. It is also remarkable the similarity of the results with the previous operation. This is due to the fact that both operations have one scalar multiplication as the main calculation and, as previously discussed, it is the main computation and it determines the performance of the operation.

4.3.3 ECDSA

The last evaluated operations are corresponding to the Elliptic Curve Digital Signature Algorithm. As explained in the 3 chapter, this algorithm is applied to the message digest of the signature (in this implementation SHA-256) which makes the ECDSA algorithm totally independent of the message size after the digest has been calculated. In this case, at least two different nodes are again involved in the operation, one of them performs the signature operation for the message and then any other node that is in possession of the public key of the signatory can verify
the authenticity and integrity of the message. In this setup, we use a fixed message "Hello Bob" that is signed. Afterwards, the message together with the signature are provided and verified.

The results corresponding to the computational time to generate the signature and verify it using the different key sizes and coordinate systems are presented if Figure 4.4.

![ECDSA generation and verification](image.png)

Figure 4.4: Computational time for ECDSA signature generation and verification for different key sizes and using the 3 different implemented coordinate systems.

It can be observed in figure 4.4 that the verification time is roughly twice the generation time. This can be explained by the fact that in the verification process two scalar multiplication operations must be performed and, as discussed earlier in 2.2.2, this is the main resource consuming operation that predominates in all the operations.

### 4.3.4 Memory usage

To conclude this evaluation chapter, we discuss the memory usage of the library using as parameters the same as in the previous sections: key size and coordinate system. As stated in the set-up section, we used the size utility for ARM compilers (arm-none-eabi-size) to obtain the flash memory size. This tool was used on the
output files of the compilation of the libraries (ECC and Bigint) to obtain the flash memory usage. As also mentioned in the evaluation set-up, we determined the RAM stack usage by writing a pattern on the RAM before the execution of the application and the corresponding scan after 10 rounds of execution to obtain the maximum used stack size. The stack usage was analyzed after the performance of all presented operations to show a full use of the library. Following we present the results in tables 4.1 and 4.2.

<table>
<thead>
<tr>
<th></th>
<th>160 bits</th>
<th>192 bits</th>
<th>256 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint.o</td>
<td>4186</td>
<td>4186</td>
<td>4186</td>
</tr>
<tr>
<td>ecc.o</td>
<td>5373</td>
<td>5361</td>
<td>5403</td>
</tr>
</tbody>
</table>

Table 4.1: Flash memory usage in bytes for different key sizes.

In Table 4.1 can be seen that the flash memory only has a different value on the ecc.o file. This is due to the fact that the only difference is the included file containing the curve parameters, which has a small increment as the key size increments since the curve parameters have the same size as the key but the proportion is very small compared to the total flash size.

<table>
<thead>
<tr>
<th></th>
<th>160 bits</th>
<th>192 bits</th>
<th>256 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Affine coordinates</td>
<td>1504</td>
<td>1652</td>
<td>1904</td>
</tr>
<tr>
<td>Homogeneous coordinates</td>
<td>1464</td>
<td>1592</td>
<td>1852</td>
</tr>
<tr>
<td>Jacobian coordinates</td>
<td>1400</td>
<td>1544</td>
<td>1796</td>
</tr>
</tbody>
</table>

Table 4.2: RAM stack memory usage in bytes in function of the key size and coordinate system.

In Table 4.2 can be seen the different RAM size used in function of the key size and the coordinate system used. It can be clearly seen that there is a relationship between the increase of the key size and the increase of stack memory usage. This corresponds to the fact of using bigger variables to represent the points in the curves and perform all the ECC operations and was an expected behavior. When the relationship between memory usage and coordinate system is analyzed, we observe that the usage of the algorithms using jacobian coordinates derives in the smallest memory usage. Even though the nature of this coordinate system could suggest otherwise when compared to affine coordinates (as previously explained, affine points are represented with two coordinates whereas homogeneous and jacobian are represented with three different coordinates), in our implementation the usage of the jacobian coordinate system results in the minimum stack memory.
usage. This is, with high probability, due to the optimization and design of the latest against the other systems.

Regarding the static RAM memory allocation the library is written using RAM stack memory only. The static variables were only used in the test application and contained the results of the operations. The variables containing the keys have obviously the same size as the key size. A special structure is the one containing the curve parameters, which, as explained in chapter 3, has a size again depending on the key size. This variable contains the values read from the aforementioned file containing the curve parameters and saved in flash. As an improvement, and if it fits the target system better, this variable could be stored in flash memory if the flash requirements are more relaxed or directly read from the file.

### 4.4 Energy consumption

To analyze the energy consumption of the implemented solution we use the obtained computational times presented in the previous sections and the data for the CPU consumption provided in the data-sheet of the used STM32W108CC chip. Since the energy consumption can be directly derived from the computational times presented in the previous sections and the consumption of the chip in CPU time, we present in this section only the consumption for the jacobian coordinate system, with a key size of 256 bits for being a significant case.

From the datasheet, with the following conditions:

- $25^\circ \text{C}$,
- $1.8 \text{ V memory and } 1.25 \text{ V core}$,
- ARM® Cortex-M3 running at 24 MHz from crystal oscillator,
- radio and all peripherals off,

the drained current by the CPU is

$$I = 7.5mA.$$

To calculate the energy consumption we use a basic linear energy model. Considering the CPU time obtained in the previous sections, the total energy consumption $E$, being $I$ the current drained, $t$ the time that the operation takes and $V_{\text{supply}}$ the supplied voltage (we take the higher value from the specifications, $1.8 \text{ V}$), is

$$E = (I \ast t) \ast V_{\text{supply}}$$
In table 4.3 the energy consumption for jacobian coordinate system, using key size of 256 bits is shown.

<table>
<thead>
<tr>
<th>Operation</th>
<th>E (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-pair generation</td>
<td>75.92</td>
</tr>
<tr>
<td>DH secret generation</td>
<td>82.02</td>
</tr>
<tr>
<td>Signature generation</td>
<td>76.23</td>
</tr>
<tr>
<td>Signature verification</td>
<td>153.84</td>
</tr>
</tbody>
</table>

Table 4.3: Energy consumption in mJ for the main ECC operations in jacobian coordinate system with a key size of 256 bits.

4.5 Conclusions

The evaluation that we just presented provides us with a good insight on the performance of the implemented ECC library and its various options regarding key size and coordinate systems used. The first clear conclusion that can be obtain is that, with the algorithms used and in the platform used for this evaluation, the jacobian coordinate system should be the used coordinate system for the ECC operations. The main reason is the better performance when compared to the other two coordinate systems in this work. Also, it is important to remark that this performance does not come with the cost of more memory usage, furthermore, the memory usage is inferior when this coordinate system is used.

When the influence of the key size is analyzed, it becomes clear that the increase of the key size translates into longer operations and an increase of memory usage. This corresponds our expectations since it was already known that all the operations manage curve points of the same size as the key. Also the increase of memory usage is due to the fact that we need to manipulate these variables of bigger size.

To conclude this evaluation section, we present in Figure 4.5 the total computational time that would require to obtain a shared secret between to entities using the ECC key-pair generation and following the ECCDH algorithm for both parties to obtain a shared secret that can be derived into a symmetric key for the establishment of a secure communication channel.
Figure 4.5: Computational time for EC key-pair generation, ECDH public key exchange and ECDH shared secret generation for different key sizes and using the jacobian coordinate system.
Chapter 5

Conclusions

5.1 Conclusion

In this work we have realized an implementation of an ECC library for the Contiki OS and thorough evaluation of this implementation in a real hardware for a common wireless sensor network device. This work shows that it is feasible and useful to use Elliptic Curve Cryptography in Wireless Sensor Networks and that its usage can be achieved with very few resources and without a critical compromise in the performance.

This work has also shown that some implementations achieve a much better performance using the same amount of resources. In this case, we have seen that the usage of the jacobian coordinate system for the platform used in this work derives in a huge boost in the performance of the operations without supposing an increase of the memory used.

We can conclude that ECC stands as an strong candidate for public-key cryptography for the Internet of Things as the computational power increases and key sizes get bigger and bigger in other public-key systems such as RSA, moving towards unmanageable key sizes.

5.2 Future work

This work implements the basic operations for the usage of the Elliptic Curve Cryptography. An immediate future work could be to incorporate the ECC primitives into an existing protocol such as IKE (Internet Key Exchange Protocol) or CoAP (Constrained Application Protocol) more in the IoT context. The library is a generic ECC library that can be used for any protocol, so the next natural step would be to integrate the solution into an existing protocol to make use of the library.
There has also been some studies about the usage of certificates with Elliptic Curves and the generation of certificates based on Elliptic Curves. This could also be an interesting field for future work. As it has also been seen in this work, the selection of algorithms, also in the underlying Bigint library has a big impact in the overall performance of the ECC operations. Future work could also be on the research for more efficient algorithms to improve the overall performance.
Bibliography


Appendix A

How to run

The work of this thesis has been performed using The Contiki OS 2.7. The libraries that we created can be used directly in a clean Contiki installation due to the fact that they are provided as a Contiki app. Simply by including the ECC library in the Contiki apps folder it can be used by any application running in Contiki. In this work we provide also an example application that uses the library and that has been used for the evaluation of the library and allows to test the library out of the box. Once the ecc app has been placed in the Contiki apps folder and the ecc-test application is placed in the examples folder, it can be run right away in a native platform.

To be able to run the application in the platform used in this work (STM32W108CC chip) a few more steps are necessary. The first step is to obtain the proper compiler for this platform. For the evaluation and for the steps here described the compiler used is the following: arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 4.8.3 20140228 (release) [ARM/embedded-4.8-branch revision 208322] Copyright (C) 2013 Free Software Foundation, Inc. which can be obtained in the GNU Tools for ARM Embedded Processors project page (https://launchpad.net/gcc-arm-embedded/4.8/4.8-2014-q1-update).

For the code to be compiled correctly, a fix needs to be done in the linker for the stm32w108 processor in the mentioned version. The following line needs to be added in the file contiki-2.7/cpu/stm32w108/gnu-stm32w108.ld:

```c
__exidx_start = .;
.ARM.exidx : { *(.ARM.exidx* .gnu.linkonce.armexidx.*) }
__exidx_end = .;
```

The next step is to ensure that there is enough stack memory. In the provided version (Contiki 2.7), the stack has a fixed size of 0x500 bytes. To ensure a correct performance of the library it is recommended to increase the size of the stack.
To do this, the value can be modified in the same linker file. In our tests and evaluation the value used was

```c
__Stack_Size = 0xA00 ;
```

To run the example application, it is only necessary to, from the ecc-test directory in the examples, compile the application for the desired platform and execute the output. In our set-up, Using the STM32W108CC the command to compile and write the program into the board is the following:

```bash
make TARGET=mbxxx STM32W_CPUREV=CC ecc-test.upload
```

This will compile the code using the mbxxx platform, which is the name for this platform in the Contiki source, and will upload the binary file to the board to run. The used RFC-kit allows the usage of the USB interface to visualize the output. To do obtain the data the following command can be used:

```bash
make TARGET=mbxxx STM32W_CPUREV=CC login
```

As mentioned previously, to run the example application in a native platform, the following command can be used to compile:

```bash
make ecc-test
```

since the Contiki OS selects the native platform as default if no other target is indicated.

As explained in the implementation chapter, the library offers the possibility to modify some of the internal parameters. For example, the size of the basic used words, the precision of the arithmetic, the coordinate system used for the ECC operations, etc. All this parameters can be modified in the Makefile of the ECC app using compilation flags.

**Coordinate system:**

- AFFINE_COORDINATES
- HOMOGENEOUS_COORDINATES
- JACOBIAN_COORDINATES

**Word size:**

- WORDS_32_BITS
- WORDS_16_BITS
Arithmetic with precision $X$:

- **NUMWORDS** = $X$

To check the used stack memory in the mbxxx platform there is also a compilation flag in the Makefile of the example test application

- **MBXXX_CHECK_STACK_USAGE**

The last configuration option is the selection of the used Elliptic Curve. The file containing the parameters just needs to be included in the application. A number of curve parameters are included in the app folder of the library.