DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2017

Digital Certificates for the Internet of Things

FILIP FORSBY

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING

Abstract

This thesis will investigate the possibility of developing a lightweight digital certificate solution for resource constrained embedded systems in 6LoWPAN networks. Such systems are battery powered or energy harvesting devices where it is crucial that energy consumption and memory footprints are as minimal as possible. Current solutions for digital certificates are found to be more demanding than what is desirable and therefore an issue that needs to be solved.

The solution that is proposed in this thesis is a profile for the X.509 certificate standard for use with constrained devices and the Internet of Things (IoT). Furthermore, a compression mechanism is designed and implemented for certificates following this X.509 profile.

Results show that compressing certificates is a highly viable solution, despite the added complexity it brings.

This new lightweight digital certificate solution will allow resource constrained systems to be able to run for longer without being interrupted or needing maintenance.

Keywords

Cyber security, constrained devices, digital certificates, Contiki OS, internet of things, embedded systems.

Sammanfattning

Denna avhandling undersöker möjligheten att utveckla lättviktslösning för inbyggda system med begränsade resurser i 6LoWPAN-nätverk. Eneheter i sådanna system drivs på batteri och återvunnen energi från omgivningen där minimal energi- och minnesanvänding är avgörande. Nuvarande lösningar för digitala certifikat anses vara mer krävande än önskvärt och det är därför ett problem som behöver lösas.

Lösningen som presenteras i denna avhandling är en profil för certifikatstandarden X.509 för användning med begränsade enheter inom Internet of Things (IoT). Utöver det är en komprimeringsmekanism designad och implementerad för certifikat som följer denna X.509-profil.

Resultat visar att det är högst genomförbart att komprimera certifikat, trots den ökade komoplexiteten det medför.

Denna nya lösning för digitala certifikat tillåter resursbegränsade enheter att köras längre utan att behöva avbrytas eller underhållas.

Nyckelord

Cybersäkerthet, begränsade enheter, digitala certifikat, Contiki OS, internet of things, inbyggda system.

Acknowledgements

Firstly I would like to thank Shahid Raza at RISE SICS for being my supervisor and helping me throughout the thesis. I would also like to thank my examiner Panos Papadimitratos at KTH for assessing and grading, and to Oskar Lundh for opposing my work. Additionally, the rest of the staff at SICS, foremost Joel Höglund and Zhitao He, deserves my gratitude for helping my with the technical difficulties I have had.

To my fellow friends and study mates, especially Martin Person and Alexander Östman, who have stood by my side from the first day at KTH until the last; I could not have done this by my self, and the help and discussions we have had through the years are invaluable.

Finally, the most sincere thanks to my family for supporting, encouraging and believing in me this whole time.

Filip Forsby
Stockholm, June 2017

Contents

Chapter 1

Introduction

This thesis is aimed to solve the problem of heavy digital certificates in the domain of Internet of Things (IoT). This domain is characterized by wireless low energy and battery powered devices, with network links with high packet loss and where all types of communication is expensive, concerning power consumption.

The solution proposed in this thesis consists of two parts, where the first is an X.509 Profile for IoT which tells how a certificate shall look like when communicating with IoT devices. Certificates following this profile will be fully valid X.509 certificates, but all existing certificates can not be used with devices conforming this profile. The second part i a compression mechanism for certificates following the profile which is applied within the 6LoWPAN network to further reduce its size.

1.1 Problem

Even though conventional certificate standards, such as the most established X.509 standard, can fit into state-of-the-art IoT systems, it is more resource demanding than what is desired. Conventional certificate standards are developed with devices like workstations and servers in mind, where factors like computational power, memory footprint and energy consumption are not main concerns. However, in battery powered and energy harvesting IoT devices, these factors are crucial and it is therefore called for to redesign these standards to be more suitable for IoT.

1.2 Purpose

The purpose of this thesis is to investigate and propose a lightweight implementation of a certificate management solution with properties such as low memory footprint, low computational complexity and minimized data transfer as the main concerns.

1.3 Goal

The goal is to design and implement a solution that is more lightweight than currently existing solutions. The design should comply with existing standards in order to be more easily acceptable by the community.

The solution will be implemented and evaluated on the Contiki OS on real hardware.

1.4 Methodology

The research strategy will be both qualitative when exploring a new design for a digital certificate, as well as quantitative when evaluating its performance. The design phase will be characterized by empirical and analytical methods with a case study on current certificates and decisions concluded after an exploratory research on alternative solutions. The evaluation is a deductive approach with experiments and a statistical analysis to compile a result.

1.5 Ethics and sustainability

This is a sustainable proposal, since the ones who will benefit from this will have longer lasting battery powered devices with possibility to lower hardware cost. In the same time, there will be no negative side effects, other than the need of deploying a new implementation for certificate authorities and clients.

Ethical problems can arise if the proposed certificate comes out insecure and confidential information will leak, or entities will be able to impersonate others, leading to information being sent to adversaries or adversaries sending malicious information to unknowing devices. The risk of this ethical problem will be minimized by using proven secure solutions as a baseline with only well reasoned modifications.

1.6 Delimitations

This thesis will focus on the design of a lightweight digital certificate solution and perform a small evaluation in performance to prove that its design goals have been met. The evaluation will not be as thorough as needed to definitely tell how much better the proposed solution will be in all kinds of environments and use cases.

This thesis focuses on certificates alone, and not the whole Public Key Infrastructure (PKI). Certificate chains are not discussed in itself, but this work is generally applicable to them as well.

1.7 Thesis outline

This thesis will start with background description in chapter 2 which will act as a baseline for the thesis. In chapter 3 there is a description of how the proposed X.509 Profile for IoT digital certificate is designed and why this particular design was chosen. Certificates following this profile can be further compressed with the compression algorithm described in chapter 4. chapter 5 describes how the compression algorithm was implemented on the Contiki OS and chapter 6 contains an evaluation of the implementation. The result of the evaluation is presented in chapter 7 and the conclusions can be found in chapter 8.

Chapter 2

Background and related work

2.1 Internet Protocol version 6

Internet Protocol version 6 (IPv6) [1] was developed to, among others, eliminate the problem of the current de facto standard IPv4, where the number of network addresses will run out. For IoT, IPv6 is an pure essential part for its existence, since the number of IPv4 addresses could never satisfy the need when every device will be connected to the Internet. An IPv6 address consists of 128 bits, compared to the IPv4 32 bits, which in theory enables 2^{128} (340 undecillion, or $3.4 * 10^{38}$) connected devices with their own IP addresses. However, in reality the number is a bit smaller as the address is divided in two with first half network address and second half device, giving 2^{64} ($1.8 * 10^{19}$) networks with 2^{64} ($1.8 * 10^{19}$) devices in each. Since all networks will probably not be filled, and all networks not used, the full address space will not be utilized, but the number of possible devices is still considered high enough. The device part of the address, the Extended Unique Identifier (EUI), is derived from the MAC address of the device. The MAC itself is also divided into two; the Organizationally Unique Identifier (OUI) which is unique for all manufacturers and the Network Interface Controller (NIC) which is device specific within the manufacturer organization. Since some MACs are only 48 bits long (for example Ethernet MACs), there needs to be something filling up the last 16 bits of the EUI. In that case, the 16 bits 0xFFFE is inserted between the OUI and NIC to form the 64 bit EUI. See Figure 2.1 for a visual representation of the IPv6 address.

2.2 6LoWPAN

IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [2] is a transmission protocol that enables IPv6 communication over the IEEE 802.15.4 wireless link protocol.

Since the synchronous Hyper Text Transfer Protocol (HTTP) is built for TCP, which is not optimal for the lossy, UDP-based IoT networks, the protocol has been

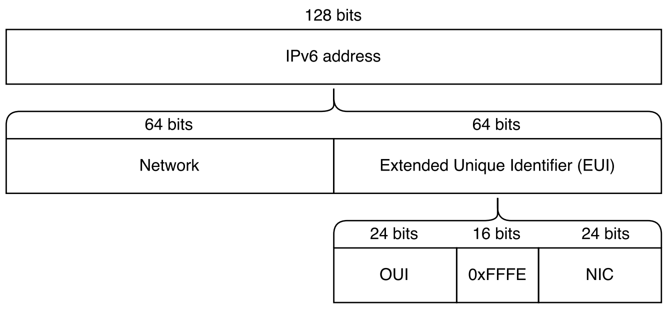

Figure 2.1. The structure of an IPv6 address.

reduced and standardized as the Constrained Application Protocol (CoAP) [3].

To secure the CoAP protocol, a protocol based on the conventional Transport Layer Security (TLS) [4], named Datagram TLS (DTLS) [5] has been developed. CoAP together with DTLS is called CoAP secure (CoAPs), in the same way as TLS with HTTP makes HTTPs, which is todays standard for securing and encrypting regular web traffic.

For this thesis it is relevant that CoAP has some restrictions on how the certificates must be constructed. Section 9.1.3.3 of the CoAP specification specifies things such as which cipher suits and subject names to be used, which has to be taken into consideration when designing a certificate for IoT.

2.3 X.509 certificates

The X.509 [6] certificate standard has been around for a long time and it is the standard used in TLS and DTLS encrypted traffic. An X.509 certificate essentially consists of three parts;

(i) information about the subject, issuer and details about the certificate such as serial number and validity dates,

(ii) the public key of the subject and its cryptographic algorithm,

(iii) a signature from the issuing Certificate Authority (CA).

As of today, there are three versions of X.509, where the latest is simply called X.509 version 3 (X.509 v3) [7]. The latest version has opened up for optional extensions, which can be marked as critical and thus has to be processed by the receiver, or is otherwise rejected. This means that these certificates are quite versatile and can be extended as the specific communities or applications require.

Figure 2.2 shows the structure of an X.509 certificate. Specific for version 3 is the extensions field, and the issuerUniqueID as well as subjectUniqueID was added in version 2. The extensions are a sequence of ID/Value pairs, with an extra critical bit.

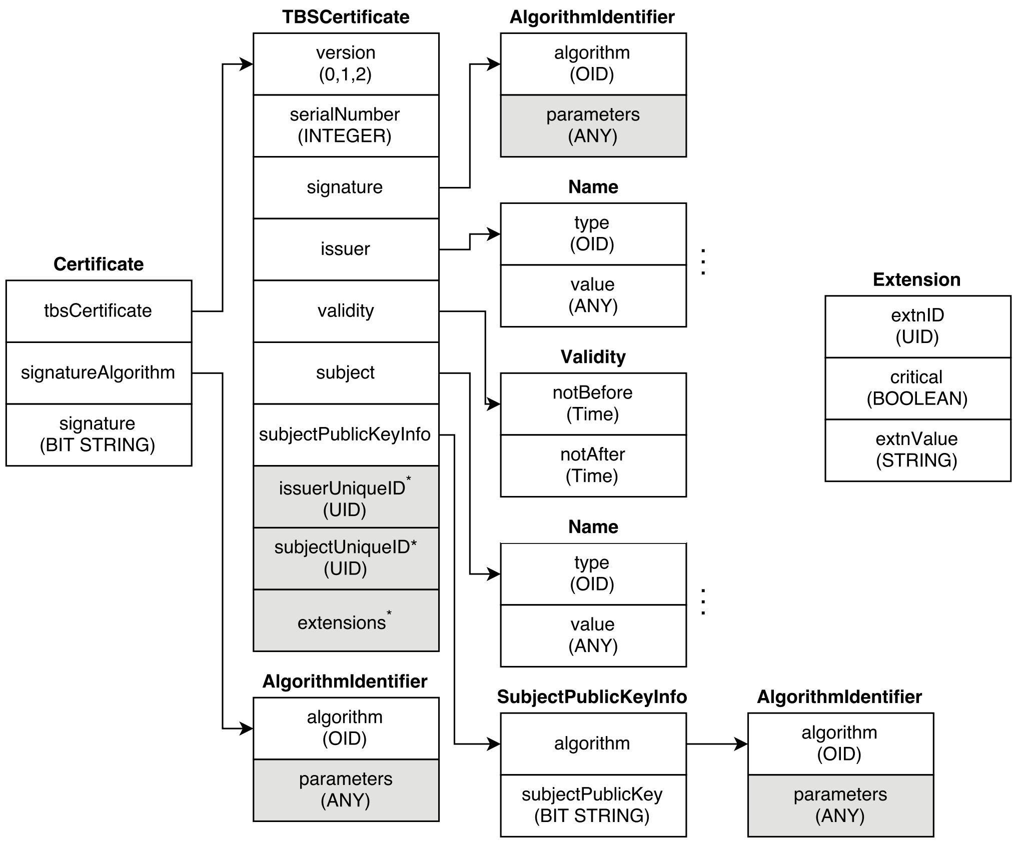

Figure 2.2. Structure of an X.509 certificate. Gray fields are optional, fields marked with star are only available in version 2 or 3 and structures with three dots can be a set of that structure.

An X.509 certificate is specified and encoded using the Abstract Syntax Notation One (ASN.1) [8] Distinguished Encoding Rules (DER), and then converted to Base64 before it is stored or sent away.

2.4 CBOR and CDDL

There was a demand on the market for a lightweight structure encoding scheme with support for binary data. ASN.1 is used for conventional X.509 certificates, but it was deemed not lightweight enough. For this the Concise Binary Object Representation (CBOR) [9] data format was invented. CBOR is designed to be extremely lightweight in terms of code size and with a small encoded message size.

The specification notation for CBOR comes from the JavaScript Object Notation (JSON) [10] and fully supports the JSON syntax and data types available in JSON. JSON is used because it is a widely accepted format and there is support and parsers for most programming languages. JSON is also both compact, which is in line with the CBOR design goals, and easy to read and interpret for both humans as well as for computers.

Even though CBOR does not rely on a specific schema in order to encode and decode messages, the CBOR Data Definition Language (CDDL) [11] was specified in order to describe and constrain CBOR structures. In the context of digital certificates, this is important in order to specify the structure of the certificates.

2.5 Contiki OS

The Contiki OS [12] is an open source operating system for embedded and constrained devices, such as the ones you would find in the Internet of Things. Contiki supports all the network protocols one could expect in the IoT, both the regular such as TCP, UDP HTTP as well as the lighter 6LoWPAN, CoAP and RPL standards. Despite all these functionalities, Contiki is very lightweight with a small memory footprint and power awareness functionality to save energy. This is a requirement for low powered and constrained devices to be able to run for a very long time only powered by batteries.

Contiki comes with a development environment containing everything from the compiler to code examples and a complete network simulator called Cooja. In Cooja, developers can set up large wireless networks of a range of different hardware and in detail debug and evaluate their applications.

2.6 Related work

There are many proposals of solutions to make secure communication in 6LoWPAN networks more lightweight. Some of them, which are connected to this thesis, are discussed in this section.

2.6.1 Raza et al. — Multiple contributions

Raza et al. have found ways to lighten both the IPSec and the CoAPs protocols for use in 6LoWPAN networks.

Compressed IPsec

In the paper *Securing Communication in 6LoWPAN with Compressed IPsec* [13], Raza et.al. design, implement and evaluate the first compressed lightweight specification of IPsec for 6LoWPAN. It includes both IPsec's Authentication Header (AH) and Encapsulation Security Payload (ESP) and provides End-to-End (E2E) secure communication between 6LoWPAN devices and generic traditional Internet hosts.

Lightweight CoAPs

Raza et.al. describe how the overhead for DTLS, which is used to secure CoAP, can be significantly reduced in their paper *Lithe: Lightweight Secure CoAP for the Internet of Things* [14]. They use the standardized 6LoWPAN mechanisms and show that the compression is efficient both in terms of energy consumption and response times.

2.6.2 Pritikin et al. — Compressed X.509 Format (CXF)

When it comes to digital certificates, there is a successful attempt [15] to compress X.509 certificates, without breaking the compatibility. Pritikin et al. use conventional compressing methods and dictionaries with reoccurring and frequently used text strings to compress X.509 certificates.

A modified version of *gzip* uses the DEFLATE [16] compression algorithm with a dictionary consisting of a typical certificate with unpopulated cryptographic fields.

The results show that the X.509 certificates they used as a test could be compressed to a rate of 0,86 for the RSA certificate and 0,73 for the ECC certificate, computed as compressed size divided by original size. The particular RSA certificate in question of 753 bytes was divided into 557 bytes of cryptographic data, 43 bytes of text and numbers, 42 bytes of Object Identifiers (OIDs) and 111 bytes structural data. With the incompressible nature of the cryptographic data, which stand for 0,74 of the total certificate size, these results are remarkable.

The blogger Graham Edgecombe has recently extended the CXF certificate [17] by constructing a new dictionary from a sample of 100'000 certificates. This dictionary turned out to reduce the size of the certificates by an additional 14% compared to the CXF dictionary. The results Edgecombe produced differ somewhat from the experiments from the CXF specification, with an even higher compression rate. The main reason for that might be that Edgecombe uses real authentic certificates while the CXF specification experiment uses one self made dummy certificate. This fact makes the CXF approach even more viable as it is the real life performance that is of interest.

2.6.3 DTLS Profiles for the Internet of Things

The Internet Engineering Task Force (IETF) has proposed a standard profile for the usage of DTLS in IoT in their *DTLS Profiles for the Internet of Things* [18]. It is a complete approach on how to use the DTLS protocol in a lightweight manner for IoT and constrained devices. Included in this profile is also a specification on certificates and their contents, where restrictions have been made to keep the certificates small and the processing of them light. This specification will in many ways be similar to the one specified in this thesis, as the goals and field of applications will be the same. This thesis will however develop the certificates even further but keep most of the functionality of the DTLS Profiles.

2.6.4 Other

Wander et.al. have made a comparison between the ECC and RSA cryptographic methods [19] on 8-bit energy-constrained platforms. The results show that ECC is signficantly better than RSA both in terms of computation time and the amount of data transmitted and stored. It is shown that the ECC cryptographic method is a highly viable solution for constrained devices.

Chapter 3

X.509 Profile for IoT

This chapter lists all the fields in an X.509 certificate and specifies its contents in both the regular X.509 specification and the proposed X.509 profile for IoT. For all fields the decided content is argued for in a discussion. The discussion is very much connected to the current IoT standards and profiles to meet their requirements. The DTLS Profiles for IoT [18] mentioned above specifies in section 4.4 the elements of the certificates, which has a big impact on the decisions for this profile.

Certificates conforming to this profile will be fully valid X.509 certificates and can be processed by any entity that can process regular X.509 certificates. However, all X.509 certificates are not processable by entities with only support for this profile, and this profile does not convert certificates between one format or another. Certificates for IoT devices have to be explicitly issued with this profile in mind, and devices that already have certificates not conforming to this profile will have to have new certificates issued.

3.1 Version

X.509 specification: An integer 0–2 specifying the version of the certificate.

Discussion: Version 3 of the X.509 standard opened up for extensions to be used, of which many are useful and serve a great purpose. Most of the certificates issued today are of version 3, since there is no real drawback compared to the earlier versions. The extended fields in version 3 are optional and certificates of version 3 does not have to be larger than previous versions certificates. Version 3 is also the only valid version when conforming to he DTLS Profiles for IoT.

IoT Profile: This field will be restricted to version 3 only. Certificates with a version different than 3 will be rejected. While there is no gain in size in this field, restricting the field to one value enables compressing to be done, by omitting the field completely.

3.2 Serial Number

X.509 specification: The serial number of the certificate given by the Certificate Authority (CA). All certificates issued by the same CA must have a unique serial number. The serial number together with the identity of the CA is the unique identifier of a certificate. The serial number consists of an integer, with unspecified signedness.

Discussion: A serial number is needed to distinguish a certificate issued by a CA from other certificates. Depending on the encoding scheme, the size of this field varies, but both CBOR and ASN.1 is concise in that manner that no more bytes than needed will be used.

While the X.509 specification does not specify the signedness of the integer, the DTLS Profiles for IoT specify that the integer is unsigned.

IoT Profile: Since it is hard to compress the serial number, no restriction is made on this value, other than that it must be positive. It is however recommended that conforming CAs use the smallest number available to save as many bytes as possible.

3.3 Signature

X.509 specification: Which algorithm that is used by the CA to sign the certificate. This must be the same as the signatureAlgorithm further down.

Discussion: This field will occur twice in a valid certificate, and by only that enables some room for compression. The practical need for this field even seems unnecessary. Peter Gutmann, which is a computer scientist at the University of Auckland in New Zeeland with a Ph.D. in cryptographic security architecture, has written a guide for programmers [20] on how to implement X.509 certificates in code. Regarding this field, he states in the guide “There doesn’t seem to be much use for this field”. Given that the signature algorithm used to sign the certificate is secure, there is no actual need to include it in the signature.

IoT Profile: No additional restrictions are added to this field, and therefore only follow the X.509 specification restrictions. However, in this profile the signatureAlgorithm will be restricted to one algorithm, which is discussed in section 3.10.

3.4 Issuer

X.509 specification: A non-empty sequence of name-value pairs that is used to identify the issuing CA, called Distinguished Name (DN). The standard fields of a DN that implementations must be prepared to receive are

- country
- organization
- organizational unit
- distinguished name qualifier
- state or province name
- common name
- serial number

Discussion: To be able to map a given certificate to a certain issuer is a key feature of digital certificates. The range of possibilities to identify an issuer of a certificate is extensive and not suited for constrained devices. In the DTLS Profiles for IoT, it is only specified that this field contains the common name (CN) with the name of the issuing CA. It could however be sane to set a restriction on the DN and not use any other fields than the CN since no constrained protocol relies on them.

IoT Profile: This field will be restricted to only contain a common name (CN) of the UTF8String type. The name must not be the same as for any other known CA.

3.5 Validity

X.509 specification: The time period of which the certificate is valid. Contains a sequence of two dates, where the first one is when the period begins and the other when the period ends. It is represented as a text string, in either UTC time with a trailing Z or in local time with specified time offset from UTC time.

Discussion: There are two date types in ASN.1, and therefore also in X.509.

One of them is UTCTime, which has the either format "YYMMDDhhmm[ss]Z" or "YYMMDDhhmm[ss](+|-)hhmm". UTCTime has only two characters for the year, which leads to ambiguity. To solve this, the X.509 standard states that if the year is "50" or greater, it shall be interpreted as 20th century (e.g. 1950), and under 50 the 21st century (e.g. 2049).

The other date type is GeneralizedTime. The format for GeneralizedTime is "YYYYMMDDhh[mm[ss[.fff]]](|Z|(+|-)hhmm)", which is a bit more complex with support for fractions of seconds and 4 character for year as the main difference.

The DTLS Profiles for IoT specifies that the value is expressed as UTCTime. However, the validity period is not mandated for use in devices with no source of absolute time, which is the same case for the CoAP protocol.

IoT Profile: To represent a date in this profile, the ASN.1 UTCTime is used, with the format `YYMMDDhhmmssZ`. While this format will be obsolete after the year 2049, it would be bad to break compatibility with the DTLS Profiles for IoT and since this is a much wider problem there might be a solution later on. If the certificate is used with devices with no source of absolute time, the time can be set to an arbitrary value.

3.6 Subject

X.509 specification: In the same way as the issuer, a sequence of name-value pairs (the DN) that is used to identify the entity with the given public key. If the subject is a CA it must have a non-empty DN matching the issuer. If the subject is not a CA, it must contain a non-empty DN unique for each subject entity certified by the one CA. The standard fields are the same as the ones of the issuer, see section 3.4, and are treated likewise.

Discussion: As for the issuer, the X.509 specification on the subject field is very extensive. This fields purpose is to uniquely identify an entity on the internet. This can be done in several different ways, where one is to "zoom" in on a geographical location like it is done in conventional X.509 applications. There is however already a unique identifier connected to an IoT device; its EUI-64. Using the EUI-64 as the only identifier is by design sufficient, since no EUI-64 is used for more than one device. If one want to give their device a "name" they can use the subject alternative name extension. One could also argue that an IPv6 address could be used as an identifier, but since an IP address will change over time, it is not suitable as a subject name.

What's not to forget is that a CA also need a certificate. In these cases it is not as convenient to use the EUI-64, for several reasons. One being that a CA may consist of several devices, and is not even device specific. A CA also usually includes humans that verify the identities of the entities they are issuing the certificates to. It makes therefore more sense to use the same format as the issuer field when the subject of a certificate is a CA.

To strengthen the reasoning supplied on this field, the CoAP specification states that the leftmost CN component or the identifier used in the SubjectAltName must be an EUI-64, if the subject is not a server.

IoT Profile: The subject field consists of on CN structure with either the EUI-64 if the subject is an IoT device, or the name of the CA if the subject is a CA. If the CN is an EUI-64 the basic constraints CA value must not be set to true. The CN is represented as an UTF8String, with the format `AB-CD-EF-01-23-45-67-89`, which is the format used in the IEEE Guidelines for EUI-64 [21].

3.7 Subject Public Key Info

X.509 specification: This field contains the the public key in a bit string and identifies which algorithm the key is used with.

Discussion: There are today two types of cryptographic algorithms used for public key cryptography.

RSA [22] is the first one, an algorithm that was first released in 1977 and builds upon the difficulty of factoring the product n of two large prime numbers p and q. In RSA, the keys consists of a pair of two positive integers, where the public key is (e, n), and the private key is (d, n). The ciphertext C of a message M is obtained by encrypting M with the encryption algorithm E using the public key (e, n);

$$C \equiv E(M) \equiv M^e \pmod{n} \tag{3.1}$$

The ciphertext C can then be decrypted with the decryption algorithm D using the private key (d, n) to obtain the original message M;

$$M \equiv D(C) \equiv C^d \pmod{n} \tag{3.2}$$

Elliptic Curve Cryptography (ECC) [23] is the other algorithm that is used, which came out about ten years later than RSA, and instead is based on the properties of elliptic curves. An elliptic curve over a field F is mathematically defined by the equation;

$$y^2 = x^3 + ax + b \tag{3.3}$$

where a and b are elements of F and not equal to 2 or 3. The math behind this algorithm is more complex than for RSA, but it relies on the difficulty to find a point on a specific curve, given two other points.

The private key in ECC is an integer, while the public key is a point of the curve, that is a pair of integers (x, y) with an extra information byte. The x and y of the public key are the same as the x and y in Equation 3.3 above, and the information byte tells whether the key is uncompressed, or compressed with an odd or even y. Compression of ECC keys will be further discussed in chapter 4. An uncompressed key has the information byte set to 0x04.

These two algorithms have different properties, with both having properties in favor over the other. One difference that have already been brought up is that the math of RSA is much easier to understand than the one of ECC. Other than the fact that RSA came first, this might be the reason why RSA is more widely used than ECC. Another difference is the number of bits needed for the public key for the same level of security. Table 3.1 shows the number of bits needed to reach a specific security level for the both algorithms. The third difference is that it has been shown that signing using ECC is much more efficient than signing with RSA, while verifying with RSA is more efficient than with ECC [19].

Table 3.1. Minimum number of bits needed in public keys for different security levels [24].

Security level (bits)	Key size (bits)	
	RSA	**ECC**
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

When deciding on which security level to go for, the National Institute of Standards and Technology (NIST) states 128 bit security level is acceptable 2031 and forward [25]. This means that a 256 bit ECC key, or a 3072 bit RSA key, would be secure enough for a long time ahead.

When it comes to already existing standards, the DTLS Profiles for IoT require the use of ECC keys, and recommends the curve *prime256v1* to be used (also known as *secp256r1* or *NIST P-256*) with uncompressed keys.

IoT Profile: With the knowledge from above, the only solution following the given design goal would be to restrict the cryptographic algorithm to 256 bits ECC keys from the curve prime256v1. As the current standards specifies that the key is uncompressed, so will this profile. This means that the OID id-ecPublicKey must be used together with the OID for prime256v1.

3.8 Issuer Unique ID and Subject Unique ID

X.509 specification: Only valid for version 2 or 3. Present to handle the possibility of reusing subject and/or issuer names. It is however recommended that CAs use unique names and must then not assign these fields.

Discussion: These fields are only necessary if the issuer or subject are duplicated. In this profile subjects are inherently unique, and issuers must use unique names, which makes these fields unnecessary,

IoT Profile: These fields must not be assigned in this profile.

3.9 Extensions

X.509 specification: Only valid for version 3. If present contains a sequence of one or more certificate extensions. An extension is a ID-value pair with a critical flag. If the flag is set, the field must be processed and if it can not be processed the certificate must be rejected.

There are standard extensions specified, for example:

- Key Usage — Defines what the key is to be used for. The value is an 8-bit bit string where each bit represent a usage scenario. For example, keys can be set to only be used for signature verification or data encipherment.
- Certificate Policies — Specifies in an end entity certificate under which policies the certificate has been issued and the purpose of the usage of it, and in a CA certificate limit the set of policies for certification paths that include this certificate.
- Subject Alternative Name — An addition to, or replacement of, the subject field of the certificate. It includes options such as Internet electronic mail address, DNS name, IP address and Uniform Resource Identifier (URI).
- Basic Constraints — Specifies whether the certificate subject is a CA, i.e. if its public key may be used to verify certificate signatures, and the maximum depth of valid certification path that include this certificate if the CA bit is set.
- Name Constraints — Must only appear on a CA certificate and restricts which name subtrees that can be used by the subjects. The URI domain (e.g. *.example.com) or mail address domain (e.g. *@example.com) are examples where this constrain apply.
- Policy Constraints — Can be used in certificates issued to a CA and constraints the path validation.
- Extended Key Usage — Extends the available 8 bits of the key usage extension with the possibility to define own usages for the key, or use predefined extensions, such as TLS server authentication or email protection.
- Inhibit anyPolicy — Can be used in certificates issued to a CA and indicates that the special anyPolicy OID is not considered an explicit match for other certificate policies except when it appears in an intermediate self-issued CA certificate.

The listed examples are extensions that applications conforming to certificate extensions must recognize. Beyond the standard extensions communities may use additional extensions, with the note that critical extensions should not prevent the certificates to be used in a general context.

Discussion: Extensions are used to provide extra information that can't be put anywhere else in the certificate. What information to provide is much connected to which field the certificates are used in. In the IoT domain, there are some extensions that makes more sense than others. To reduce the complexity needed for implementations, it is mandated to remove the need to implement extensions that are less needed.

In the DTLS profile for IoT, there are four extensions that must be implemented; key usage, subject alternative name, basic constraints and extended key usage. These are all basic extensions that can be applied to almost all certificates, and they all have their use in the IoT domain.

While the use of extensions will make the certificates larger, this is the opportunity to customize certificates for application specific domains. This profile is not aimed to restrict the certificates more than necessary and shall therefore not limit the allowed extensions to a certain set.

IoT Profile: Any extension is allowed in this profile. There are some extensions that entities conforming to this profile must be prepared to receive. These are:

- Key Usage
- Subject Alternative Name
- Basic Constraints
- Extended Key Usage

This makes the following extensions optional in the profile, which previously was required in the X.509 specification:

- Certificate Policies
- Name Constraints
- Policy Constraints
- Inhibit anyPolicy

It is recommended that CAs do not issue certificates with extensions other than the four first listed. Should an additional extension be needed, it is recommended to make the extension as limited as possible to keep the size down.

If the subject of the certificate is a CA, then the extension Basic Constraints must be present with the CA value set to true.

3.10 Signature Algorithm

X.509 specification: Contains the identifier for the cryptographic algorithm used by the CA to sign the certificate. This field must contain the same algorithm identifier as the signature field listed above in section 3.3.

Discussion: For the CA to create a signature for the certificate there are two steps that need to be taken; first calculate a hash of the certificate and then encrypt the hash with the private key of the CA. When validating the signature, one must know both the hashing algorithm and the encryption algorithm, since they all will generate different signatures for the same certificate. In X.509, there are a range [26] of different combinations of hashing and encryption algorithms to choose from, and the chosen combination is expressed as a unique ASN.1 Object Identifier (OID).

The DTLS Profile for IoT specifies ECDSA signature algorithm with SHA256 or stronger.

IoT Profile: There is no reason to support stronger hashing algorithms than SHA256 since it is assumed to be secure, and the use of a 256 bit ECC curve makes a longer hash pointless. ECDSA is the elliptic curve version of the Digital Signiature Algorithm (DSA), and the differences are similar to the ones of ECC and RSA. For example, an ECDSA signature produces a smaller signature and uses smaller keys than DSA. An important factor when deciding the signature algorithm is also the support from hardware. Hardware solutions that supports ECC public key cryptography are also very likely to support ECDSA signatures.

For the reasons above, the signature algorithm in this profile is restricted to ECDSA with SHA256, and thus the ASN.1 OID ecdsaWithSHA256.

3.11 Signature

X.509 specification: This field is used to verify that the information given in the certificate is correct and that the subject of the certificate is the true owner of the given public key. The input to the signature algorithm is the ASN.1 DER encoded structure tbsCertificate and it is the private key of the CA that is used to sign it. The output value from the algorithm is present in this field encoded as a bit string.

Discussion: An ECDSA signature consists of two values; r and s, which both are integers. The value r is message independent and calculated by the signer from a random value k multiplied with a point G on the curve. The value s can be calculated using the value r, k, the signer's private key and the hash of the data to sign. The signature is validated by calculating a point x, y on the curve from the values r and s from the signature together with the hash of the data and the public key of the signer. If the value x is equal to r, the signature is valid (that is, that the private key mapped to the public key was used to sign the data).

Unlike the x and y value of an ECC public key, r and s are not points on the curve and can therefore not be compressed in the same way. In fact, the whole s is used to compute r, and no easy compression can be made. There are however some patented solutions for compressing ECDSA signatures, for example *Compressed ECDSA signatures* (patent number US 8631240 B2) [27], where the s value is replaced by a smaller value c.

IoT Profile: Since the signature algorithm to be used is ECDSA, this field will contain the signature generated by the ECDSA algorithm. This signature must use the format ECDSA-Sig-Value described in RFC5480 [28]:

```
ECDSA-Sig-Value ::= SEQUENCE {
     r  INTEGER,
     s  INTEGER
}
```

The r and s values in an ECDSA signature are both 256 bits (32 bytes) unsigned integers, when using the *prime256v1* curve. Since the ASN.1 integers are signed, an extra `0x00` byte must be added as the most significant byte if the most significant bit in the 32 byte integer happens to be `1` (that is if the integer is equal to or greater than 2^{255}).

3.12 Summary

The summary of all the fields and their values can be found in Table 3.2.

Table 3.2. Summary of field contents in the X.509 Profile for IoT.

Field	**Value**
Version	3
Serial number	Unsigned integer
Signature	ecdsaWithSHA256
Issuer	CommonName containing CA name as UTF8String
Validity	UTCTime in format YYMMDDhhmmssZ
Subject	CommonName containing CA name or EUI-64 as UTF8String
Subject public key info	ecPublicKey followed by prime256v1 and 64 byte uncompressed ECC public key
Issuer and subject unique ID	Not present
Extensions	Any extension
Signature algorithm	ecdsaWithSHA256
Signature	ECDSA-Sig-Value ::= SEQUENCE {r INTEGER, s INTEGER}

The full ASN.1 specification for the X.509 Profile for IoT is attached in Appendix B.

Chapter 4

Compression

Certificates conforming to the X.509 Profile for IoT are highly qualified for being compressed. This chapter describes the techniques used and details on how to compress each field.

The compression will be made at the 6LoWPAN border gateway, and decompression in the nodes when verifying the signature, but also in the border gateway for outgoing certificates. Since certificates are exchanged in the setup phase of a secure connection and are therefore not encrypted, the border gateway can seamlessly compress the certificates when they pass.

The CDDL specification for a compressed certificate can be found in Appendix B.

4.1 Techniques

To perform the compression there are some general techniques used. This section describes these techniques and how they were used.

4.1.1 CBOR

As previously mentioned, the X.509 standard uses ASN.1 encoding for structure description. ASN.1 has been shown to not be the most lightweight of encoding structures and binary data, both when it comes encoding and decoding complexity and length of the encoded string. These are two reasons why CBOR was created; to firstly be very easy to encode and decode and secondly produce a small encoded string. Since the compressed certificate is independent from the original one in terms of encoding standard, it is called for to use a more lightweight solution. The compressed certificates are therefore encoded using the CBOR encoding schema.

4.1.2 ECC point compression

As described in section 3.7 and also mentioned in [23], an ECC public key, which is a point on a curve, can be compressed.

Compressing is done by omitting the y-value and changing the information byte, depending on the characteristics of y. The information byte will either be 0x02 or 0x03 when the key is compressed. Since the equation of the given curve is is known ($y^2 = x^3 + ax + b$), y be calculated from x as the square root of $x^3 + ax + b$. From this you get two possible y, and the information byte tells which of them is correct. The details of compression and decompression of ECC public keys can be found in [29] section 2.3.3 and 2.3.4, respectively.

4.1.3 Omitting implied fields

Fields that are fixed can easily be omitted since they are considered as implicit. In the decompression phase, the static values for these fields are inserted on the correct places.

4.1.4 Text to bytes

ASN.1 is in many ways very user friendly, and many of the structures and data types are chosen to be human readable. This is obviously not the most lightweight way of doing it, and this compression mechanism translates the human readable text to its binary counterpart. This is done in two places, where the first is the EUI-64 of a subject field. The EUI-64 is converted from a text string to a byte string with the binary values. For example, the string “FF” (two bytes) is converted to 0b1111 1111 (one byte). The other place where it is used when representing time. In ASN.1 the time is a textual string containing year, month, day, hour etc, which can easily be translated to a more compact representation. In this thesis, the UnixTime integer was chosen, which is the number of seconds since January 1st, 1970 UTC.

4.2 Version

The version number is omitted when compressing, since it is implied to be version 3. When decompressing, the following static ASN.1 code is added to the certificate.

```
-- Uncompressed ASN.1 --
0xA0    // Tag 0
0x03    // Size 3
0x02    // Integer
0x01    // Size 1
0x02    // Value v3(2)
```

4.3 Serial Number

Without knowing the method of how the serial number is derived, it is hard to compress this number. A possible way to compress is to determine a value which the serial numbers are distributed around, or some kind of mean value, and compress the serial number by giving an offset from this value. However, while this offset could be smaller than the actual number, it can also be bigger and also adds additional complexity. The serial number is therefore left uncompressed and the only action taken is to convert from ASN.1 encoding to CBOR.

An example with serial number 42:

```
-- Uncompressed ASN.1 --            -- Compressed CBOR --
0x02    // Integer                  0x18    // Integer of size 1
0x01    // Size 1                   0x2A    // Value 42
0x2A    // Value 42
                                        (2 bytes)
    (3 bytes)
```

4.4 Signature

This field is omitted when compressing the certificate. There are two main reasons why this can be omitted, where the first is that it is implied to be the same as signatureAlgorithm. The second reason is inherited from signatureAlgorithm, since that field is restricted to one value in the X.509 Profile for IoT, which is ecdsaWithSHA256.

When decompressing, this field is therefore statically assigned to the ASN.1 OID for ecdsaWithSHA256:

```
-- Uncompressed ASN.1 --
0x30    // Sequence
0x0A    // Size 10
0x06    // OID
0x08    // Size 8
0x2A 0x86 0x48 0xCE 0x3D 0x04 0x03 0x02
        // 1.2.840.10045.4.3.2
            (ecdsaWithSHA256)

    (12 bytes)
```

4.5 Issuer

In the same way as the CXF described in the related work, this field could take advantage of a dictionary of common CA names. The dictionary must however be carefully derived, and changed over time. It is also possible that the CA names will be more different in the IoT domain and the dictionary can therefore grow big and thus require much storage room. This type of design is disregarded for stated reasons and only a simpler compression is made.

This simpler compression is limited to only omit the ASN.1 OID for commonName, which can be done since it is implied by the X.509 Profile for IoT. See an example below for a ca with name “Root CA”.

```
-- Uncompressed ASN.1 --
0x30    // Sequence
0x12    // Size 18
0x31    // Set
0x10    // Size 16
0x30    // Sequence
0x0E    // Size 14
0x06    // OID
0x03    // Size 3
0x55 0x04 0x03
        // 2.5.4.3
            (commonName)
0x0C    // UTF8 string
0x07    // Size 7
0x52 0x6F 0x6F 0x74
    0x20 0x43 0x41
        // Value "Root CA"

    (20 bytes)
```

```
-- Compressed CBOR --
0x67    // Text string of size 7
0x52 0x6F 0x6F 0x74
    0x20 0x43 0x41
        // Value "Root CA"

    (8 bytes)
```

4.6 Validity

There are many different ways of expressing time and dates. The method used in the X.509 certificate is to write the year, month, day and time of day as a human readable text string, which is not the most compact way. Table 4.1 shows a comparison between different ways of expressing date and time.

Table 4.1. Bytes needed for different ways of storing date and time.

Standard	Bytes	Comment
ASN.1 UTCTime[1]	11–17	One of the alternatives in X.509
ASN.1 GeneralizedTime[2]	10–23	The other alternative in X.509
VS2015 DATE[3]	8	Using floating point representation
UnixTime[4]	4–5	Native by many systems

When it comes to size it is clear that UnixTime requires the least amount of bytes to represent a date. Four bytes are needed to express a point in time that is before January 2038, and after that an extra byte is needed, making it 5 bytes. This way of representing time is standard in many systems and uses only a signed integer which is the number of seconds since January 1st 1970.

The ASN.1 representations used in conventional X.509 certificates are the longest of them all, with up to almost six times more bytes needed than UnixTime. This format is more readable by humans than the other formats, but makes it more heavy for a computer to process.

The textual format is compressed to UnixTime for maximal efficiency. Since UTCTime is implied, the structural specifiers are omitted.

An example validity tuple is displayed below:

```
-- Uncompressed ASN.1 --                   // Value "270513124206Z"
0x30    // Sequence
0x1E    // Size 30                     (32 bytes)
0x17    // UTCTime
0x0D    // Size 13
0x31 0x37 0x30 0x35 0x31 0x32 0x30
    0x38 0x35 0x35 0x32 0x36 0x5A
        // Value "170512085526Z"
0x17    // UTCTime
0x0D    // Size 13
0x31 0x37 0x30 0x35 0x31 0x33 0x31
    0x32 0x34 0x32 0x30 0x36 0x5A
```

[1]https://www.obj-sys.com/asn1tutorial/node15.html.
[2]https://www.obj-sys.com/asn1tutorial/node14.html.
[3]https://msdn.microsoft.com/en-us/library/82ab7w69.aspx.
[4]https://en.wikipedia.org/wiki/Unix_time.

```
-- Compressed CBOR --
0x82    // Array of size 2
0x1A    // Integer of size 3
0x59 0x15 0x78 0x7E
        // Value 1494579326
0x1A    // Integer of size 3
0x59 0x16 0xFF 0x1E
        // Value 1494679326

    (11 bytes)
```

4.7 Subject

Following the X.509 Profile for IoT there are two possible formats for the CN of the subject field; either a CA name or an EUI-64.

If the CN consits of an EUI-64 it is compressed to the binary bytes that the EUI-64 represents. Since the CN is a text string, the raw byte representation will be much smaller, since a binary byte is represented as two characters of one byte each. The textual EUI-64 is also divided into four tuples separated with a “-”, which can be removed.

See below for an example of a compressed EUI-64:

```
-- Uncompressed ASN.1 --
0x30    // Sequence
0x22    // Size 34
0x31    // Set
0x20    // Size 32
0x30    // Sequence
0x1E    // Size 30
0x06    // OID
0x03    // Size 3
0x55 0x04 0x03
        // 2.5.4.3
            (commonName)
0x0C    // UTF8 string
0x17    // Size 23
0x30 0x31 0x2D 0x32 0x33 0x2D 0x34
    0x35 0x2D 0x36 0x37 0x2D 0x38
    0x39 0x2D 0x41 0x42 0x2D 0x43
    0x44 0x2D 0x45 0x46
        // Value
        "01-23-45-67-89-AB-CD-EF"

    (36 bytes)
```

```
-- Compressed CBOR --
0x48    // Byte array of size 8
0x01 0x23 0x45 0x67 0x89
    0xAB 0xCD 0xEF
        // Value
        0x0123456789ABCDEF

    (9 bytes)
```

4.8 Subject Public Key Info

Since it is implied to use ECC keys with the curve prime256v1, this information can be omitted.

The ECC public keys is compressed using the technique described previously in this chapter.

The public key field will thus consist of the compressed ECC public key.

```
-- Uncompressed ASN.1 --           -- Compressed CBOR --
0x30    // Sequence                0x58 0x21
0x59    // Size 89                         // Byte string of size 33
0x30    // Sequence                0x0* [ECC value X]
0x13    // Size 19
0x06    // OID                         (35 bytes)
0x07    // Size 7
0x2A 0x86 0x48 0xCE 0x3D           (Where * is 2 or 3, depending
    0x02 0x01                          on even or odd value y)
        // 1.2.840.10045.2.1
            (ecPublicKey)
0x06    // OID
0x08    // Size 8
0x2A 0x86 0x48 0xCE 0x3D
    0x03 0x01 0x07
        // 1.2.840.10045.3.1.7
            (prime256v1)
0x03    // Bit string
0x42    // Size 66
0x00    // Unused bits 0
0x04 [ECC value x] [ECC value y]

    (91 bytes)
```

4.9 Issuer Unique ID and Subject Unique ID

Since these fields must not be assigned in the X.509 Profile for IoT, there is nothing to compress and these fields are ignored.

4.10 Extensions

Extensions consist of three parts; an OID, a boolean telling if it is critical or not, and a ASN.1 DER encoded bit string as the value. The OIDs are compressed by omitting the first two bytes, which will always be 0x551D (2.5.29; joint-iso-itu-t(2) ds(5) certificateExtension(29)): The rest of the OID bytes are used as a tag for the CBOR structure which has the format:

```
[tag, critical*, value]
```

where `critical` is an optional true or false value, which will have the same value as the ASN.1 counterpart, or omitted if it implicit false. The value will contain the DER encoded bit string, as a compression mechanism for all possible extensions and their variants will be too complex to fit in this simple protocol.

See below for an example of a compressed extension field:

```
-- Uncompressed ASN.1 --           -- Compressed CBOR --
0xA3    // Tag 3                   0x82    // Array of size 2
0x1D    // Size 29                 0x83    // Array of size 2
0x30    // Sequence                0x13    // Value 19
0x1B    // Size 27                 0xF5    // Value true
0x30    // Sequence                0x42    // Byte string of size 2
0x0C    // Size 12                 0x30 0x00
0x06    // OID                             // Value
0x03    // Size 3                  0x82    // Array of size 2
0x55 0x1D 0x13                     0x0F    // Value 15
        // 2.5.19                  0x44    // Byte string of size 4
            (basicConstraints)     0x03 0x02 0x02 0x84
0x04    // Bit string                      // Value
0x05    // Size 5
0x30 0x03 0x01 0x01 0xFF               (14 bytes)
        // Value
0x30    // Sequence
0x0B    // Size 11
0x06    // OID
0x03    // Size 3
0x55 0x1D 0x0F
        // 2.5.15
            (keyUsage)
0x04    // Bit string
0x04    // Size 4
0x03 0x02 0x02 0x84
        // Value

    (31 bytes)
```

4.11 Signature Algorithm

This field is omitted when compressing a certificate conforming to the X.509 Profile for IoT, since the only algorithm that is allowed is ECSDA with SHA256. Therefore, when decompressing, this field is statically assigned to the ASN.1 OID for ecdsaWithSHA256, which is the same as the signature field further up.

```
-- Uncompressed ASN.1 --
0x30    // Sequence
0x0A    // Size 10
0x06    // OID
0x08    // Size 8
0x2A 0x86 0x48 0xCE 0x3D 0x04 0x03 0x02
        // 1.2.840.10045.4.3.2
            (ecdsaWithSHA256)

    (12 bytes)
```

4.12 Signature

This field can be compressed, since the ASN.1 structure contains unnecessary information on the values. It is known that the signature will consist of two unsigned integers of size 32, which is the only valuable information. By only sending 64 bytes as a byte array, the structure bytes for bit string and integers can be omitted. The first half of the array contains the r value, and the second half contains the s value.

The compressed value can be decompressed by adding the structural bytes, and if needed the 0x00 padding byte if r or s are bigger than 2^{255}.

Here is an example of a compression of a signature:

```
-- Uncompressed ASN.1 --              -- Compressed CBOR --
0x03    // Bit string                 0x58    // Byte array
0x48    // Size 72                    0x40    // Size 64
0x00    // Unused bits 0              [32 bytes r value]
0x30    // Sequence                   [32 bytes s value]
0x45    // Size 69
0x02    // Integer                        (66 bytes)
0x21    // Size 33
0x00    // Padding
[32 bytes r value]
0x02    // Integer
0x21    // Size 33
0x00    // Padding
[32 bytes s value]

    (75 bytes)
```

Chapter 5

Implementation

The compression mechanism that have been described in the previous chapter has been implemented as an app for the Contiki OS. The app supports compression, decompression, verification of compressed certificates and creation of new certificates.

The app is called *xiot* (from X.509 for IoT) and is typically placed in the `Contiki/apps/xiot` directory. All functions and types have the *xiot_* prefix to not interfere with other apps or libraries.

5.1 Structure

In this implementation, a certificate can be in three different stages:

Uncompressed The full X.509 certificate, conforming to the X.509 Profile for IoT, that has either not been compressed yet, or has been decompressed. This is what the certificate looks like outside the 6LoWPAN network. It is represented as a byte array containing the ASN.1 DER encoded structure.

Compressed The compressed version of a certificate, which has been compressed according to the rules specified in chapter 4. When a certificate travels within the 6LoWPAN network, this is what it looks like. The representation is a byte array containing the CBOR encoded structure.

Decoded When a compressed certificate arrives to the endpoint, it has to be decoded in order to be processed. The decoded version is a C struct with all the fields from the compressed certificate. This struct is used when the certificate is verified and certain fields needs to be accessed.

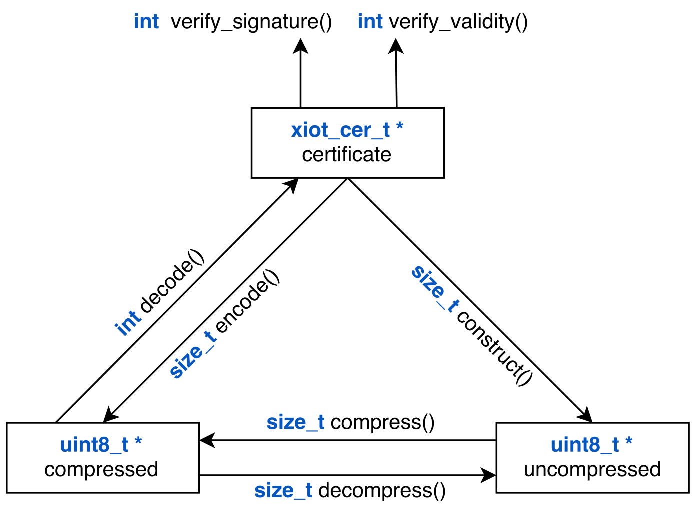

Figure 5.1. Structure of the app implemented for Contiki OS, with data types and functions.

The transitions between these stages are done with the functions in the app, which are the following:

Compress Create a compressed certificate from an uncompressed certificate.

Decompress Create an uncompressed certificate from a compressed certificate.

Decode Create a struct from a compressed certificate.

Encode Create a compressed certificate from a struct.

Construct Create an uncompressed certificate from a struct.

Two other functions are included in the app which are not a transition between stages are:

Verify_signature Takes a struct as the input, together with the CA public key, and verifies that the signature of the certificate is correct.

Verify_validity Takes a struct as the input, together with a date, and verifies that the date is within the validity period.

Figure 5.1 shows the relationships between the stages and functions in the app.

5.2 Dependencies

For the app to work, it needs a few dependencies. Since some parts of the compression mechanism is unspecific for this thesis, such as the Elliptic Curve Cryptography and CBOR encoding, it makes sense to use already existing solutions that are well known and have been proven over time. Some of this functionality already exist as other Contiki apps, while other need external support.

5.2.1 Contiki

Contiki comes with several of apps, and there are also unofficial apps written by other Contiki developers. This app uses two parts from the EST app for Contiki:

est-x509 An X.509 parser for encoding, decoding and working with X.509 certificates.

ecc A small library for working with ECC keys and cryptography.

5.2.2 External

Two additional dependencies are brought from external sources and are not in them selves connected to Contiki:

cn-cbor [1] "A constrained node implementation of CBOR in C". Used to encode and decode the compressed certificate, which is CBOR encoded.

micro-ecc [2] "ECDH and ECDSA for 8-bit, 32-bit, and 64-bit processors". Used for ECC public key creation and compression.

5.3 Usage

This section is a guide on how to use the provided implementation.

The precondition to run this code is that Contiki is downloaded and a working build environment for the target platform is set up. All other necessary files and libraries are included in the app.

To set up the environment, the script `setup.sh` is run where links are set up within the Contiki installation. For this to work, the path to the Contiki folder needs to be set correctly in the script file. If this code is run within the Instant Contiki virtual machine, the path should already be correct.

[1]https://github.com/cabo/cn-cbor
[2]https://github.com/kmackay/micro-ecc

5.3.1 Functions

All the necessary information about functions, structures and constants can be found in the file `src/xiot/xiot.h`. Here is a summary of the functions and how to use them:

Compress — Takes a pointer to a full X.509 certificate and a pointer where to store the compressed certificate. Returns the size of the compressed certificate.

```
size_t xiot_compress(
            uint8_t* compressed,
            const uint8_t* uncompressed,
            size_t length);
```

Decompress — Takes a pointer to a compressed certificate and a pointer where to store the decompressed certificate. Returns the size of the decompressed certificate.

```
size_t xiot_decompress(
            uint8_t* decompressed,
            const uint8_t* compressed,
            size_t length);
```

Construct — Takes the pointer to a structure holding a certificate and a pointer where to store the new uncompressed certificate. Returns the size of the new certificate.

```
size_t xiot_construct(
            uint8_t* decompressed,
            xiot_cert_t* cert,
            uint8_t* ca_private);
```

Encode — Takes the pointer to a structure holding a certificate and a pointer where to store the compressed certificate. Returns the size of the compressed certificate.

```
size_t xiot_encode_compressed(
            uint8_t* compressed,
            xiot_cert_t* cert);
```

Decode — Takes a pointer to a compressed certificate and to a structure where to hold the decoded certificate. Returns 1 if success, otherwise 0.

```
int xiot_decode_compressed(
            xiot_cert_t* cert,
            const uint8_t* compressed,
            size_t length);
```

Verify — Takes the pointer to a structure holding a certificate and a pointer to the public key of the issuing CA for the signature and a given time for the validity. Returns 1 if valid, otherwise 0.

```
int xiot_verify_signature(
            xiot_cert_t* cert,
            uint8_t* public_key);

int xiot_verify_validity(
            xiot_cert_t* cert,
            time_t time);
```

5.3.2 General advice

Since the RAM in these constrained devices is very limited, it is important to use static buffers and variables wherever possible. In Contiki, the *memb* API can be used for static memory allocation, or just global and static variable. In the example (see next section), certificate buffers are just statically allocated byte arrays, and keys are global byte arrays.

5.3.3 Example

Together with the implementation, there is a test program provided. This program tests all the functionality and verifies that uncompressed and decompressed certificates are the same, as well as compressed and encoded certificates. The program can be found in the folder `src/xiot_test` and it runs on the local computer if the command `make && ./xiot_test.native` is executed within the folder and the path to Contiki is set appropriately in the Makefile.

5.4 Source code

The source code of the implemented app can be found on GitHub:

https://www.github.com/not_there_yet

Chapter 6

Evaluation

An evaluation is made to determine the actual gain in power efficiency the compression mechanism yields. Since the compression introduces extra complexity, which leads to additional computation, it is not clear how much power is saved, if any at all.

As a baseline, the already existing *est-x509* app for Contiki will be used. This evaluation therefore becomes a comparison between *est-x509* and the compression mechanism described in chapter 4. In both cases the same example certificate will be used, a certificate that conforms to the X.509 Profile for IoT, which is described in chapter 3. The profile itself is thereby not evaluated.

In this evaluation, six different functions will be probed for energy consumption. These are the six functions:

Decode From a byte array containing an encoded certificate, create a matching C structure with all fields and values.

Encode From the C structure mentioned above, create an encoded byte array.

Verify Check that the signature is made using a private key matching given public key and that a given date lies within the validity period .

Compress Only for the compression mechanism. Compress a full certificate (as a byte array) and create a reduced representation (as a byte array).

Decompress Only for the compression mechanism. Take a compressed representation (as a byte array) and return it to its original representation (as a byte array).

Transmission The pure transmission of the certificate to the end node, which is either compressed or uncompressed.

To create a verdict, the results of all the functions combined will be compared between the two candidates. As the compression will not be done at the end node but on the border gateway, which is connected to a wired power supply, the compression performance is not critical. Instead, the sum of transmission, decode and verify should be compared as the main issue, since that is what will happen at the end node in the normal use case.

6.1 Platform

The evaluation is done on the Zolertia Firely[1] motes, which has the following properties:

- ARM® Cortex®-M3 TI CC2538 MCU
- Up to 32 MHz core clock
- 32 KB RAM memory
- 512 KB flash memory
- Power consumption from 7 mA at 16 MHz clock speed without peripherals and 20 mA or 24 mA with active radio in receive or transmit mode, respectively

The full specification can be read in the CC2538 datasheet [30] and on the Zolertia Firefly GitHub page [31].

6.2 Memory usage

Memory usage will be evaluated in two ways, where one is the actual size of a certificate, and the other is the size of the compiled code for the implementation. For the compiled code, the amount of data is divided into ROM and RAM usage.

Measuring the size of the certificates is simply counting the amount of bytes, while the code size needs an external program to measure what parts of the compiled code goes where. For this a platform specific tool is used, and in this case where the MCU is an ARM Cortex M3, the tool is *arm-none-eabi-size*.

[1]http://zolertia.io/product/hardware/firefly

6.3 Energy consumption

To measure power consumption in Contiki, the Energest [32] module is used. Energest can measure the time individual peripherals have been active, and with known current ant voltage levels the power can be calculated.

The time Energest returns is given in *ticks* and must be divided by the number of ticks per second to retrieve the time in seconds. The formula for calculating energy usage is therefore:

$$Energy = \frac{ticks}{ticks/second} \times Volt \times Ampere \tag{6.1}$$

To use energest, the file `energest.h` must be included and the macro `ENERGEST_CONF_ON` must be defined to 1 in the `project-conf.h` file (which is included in the Makefile).

To get the amount of time the CPU has been active, the function `energest_type_time()` is called with the argument `ENERGEST_TYPE_CPU`. The return value is an unsigned long integer.

To do time measurement for some specific operations, the following snippet will make sure that `cpu_time` holds the amount of ticks the CPU spent active while performing the operations:

```
unsigned long cpu_start, cpu_time;
cpu_start = energest_type_time(ENERGEST_TYPE_CPU);
/* Do operations */
cpu_time = energest_type_time(ENERGEST_TYPE_CPU) - cpu_start;
```

6.4 Compatibility

To prove that certificates generated with this implementation are valid X.509 certificates, the certificates are parsed with OpenSSL[2], which is “an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols”. If the certificates pass the parsing without any errors, they are considered as X.509 compatible. For parsing an X.509 certificate with the name `certificate.crt`, the OpenSSL command used is the following:

```
$ openssl x509 -in certificate.crt -text -noout
```

[2]https://www.openssl.org/

Chapter 7

Results

7.1 Memory usage

There are three different kind certificates that have been compared; a regular X.509 certificate, a certificate conforming to the X.509 Profile for IoT and a compressed version of the same profiled certificate. The sizes of these three certificates are shown in Figure 7.1. In this case, the regular X.509 certificate is a generic example taken from the Internet web page *https://www.example.com*. The profiled certificate is an example with an EUI-64 as subject and with 2 extensions. All these certificate can be found in Appendix A

Both the profiled certificate and the example.com certificate are base64 encoded surrounded with the strings -----*BEGIN CERTIFICATE*----- and -----*END CERTIFICATE*-----, while the compressed certificate is pure binary. This means that the actual information the first two is slightly less, both because of the extra

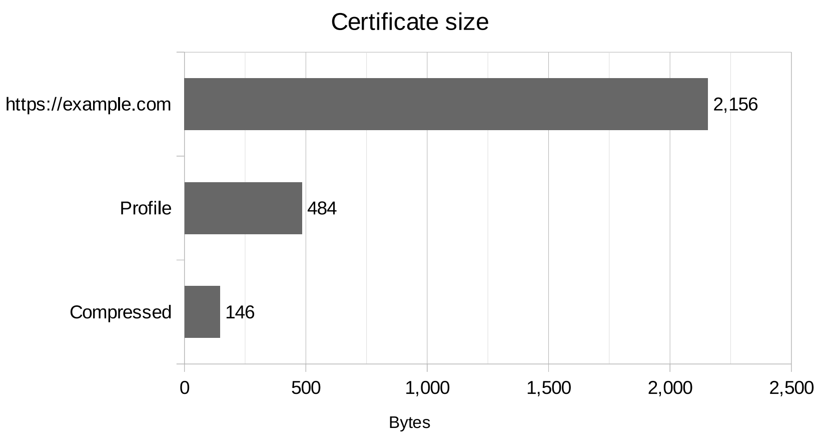

Figure 7.1. The size in bytes for different example certificates.

52 bytes of text and the base64 encoding. Base64 encoding increases the size by one third, or by ~ 33%, this because it takes 3 bytes and converts them to four printable characters, which is 1 byte each. The last one or two characters might be the "=" character, if the input string is not an even multiple of 3 bytes. That is if the input size $\equiv 1 \pmod 3$, there are 2 "=" characters and if it the size $\equiv 2 \pmod 3$, there is 1 "=" character, since the "=" characters express how many bytes are missing from a full triplet.

To calculate the compression ratio of a certificate, the binary size first has to be calculated. In this case there is no "=" characters, so the size would be calculated as:

$$\frac{(484 - 52) * 6}{8} - 0 = 324 \tag{7.1}$$

The ratio can then simply be calculated as:

$$\frac{324}{146} \approx 2.22 \tag{7.2}$$

Thus, the compression ratio for the given example is slightly greater than 2:1.

The size of a certificate can be broken down into sizes of individual fields. In Table 7.1, the field sizes for the different types of certificates are shown. These are the binary sizes without base64 encoding, and the total sizes are therefore less than what is shown in Figure 7.1. This visualizes where the most bytes are saved and where compression does not do much difference.

It shows for example that the *subject* and *issuer* fields are greatly reduced with the profile, and so are the cryptographic parts and the extensions. These are also the fields that leverage the most from compressing.

The fields *version, serial number, signature, validity* and *signature algorithm* are very little affected by the profile, if any at all.

When it comes to the size of the compiled program, the different memory areas are shown in Table 7.2. It is clear that adding compression mechanisms on top of the regular library adds extra size. The size added is about 1.3 kB on the text area, 0.8 kB on the data area and 2 kB on the bss area.

7.2 Energy consumption

The amount of energy each part of certificate handling consumes has been measured, and the results are shown in Table 7.3. When no hardware support for ECC operations, the verification step is by far the most dominant consumer. In this case, the gain from smaller size is not as evident as when hardware cryptography is used.

Without hardware cryptography, the uncompressed certificate consumes ~2.2 % more energy than the compressed, whereas with hardware becomes as much as ~23.4 %.

Table 7.1. Size of individual fields for different certificates.

Field	**Field size (Bytes)**		
	No profile	**Uncompressed**	**Compressed**
Overhead	8	7	1
Version	5	5	0
Serial number	18	3	2
Signature	15	12	0
Issuer	114	20	8
Validity	32	32	11
Subject	168	36	9
Subject public key info	294	91	35
Issuer and subject unique ID	0	0	0
Extensions	596	31	14
Signature Algorithm	15	12	0
Signature	261	75	66
Total	**1526**	**324**	**146**

Table 7.2. Size comparison of compiled program and memory areas for compressed (xiot) and uncompressed (est).

Version	**Area size (Bytes)**		
	Text	**Data**	**BSS**
est	30779	1710	7009
xiot	44081	2542	9053

Table 7.3. Energy consumption for different operations with uncompressed and compressed certificates.

Operation	**Energy consumption (mJ)**	
	Uncompressed	**Compressed**
Receive	6.55	4.69
Transmit	23.20	15.11
Decode	0.01	1.26
Verify (SW)	305.00	306.40
Verify (HW)	16.13	16.14
Total		
Software	**334.76**	**327.46**
Hardware	**45.89**	**37.20**

In Figure 7.2, a visualization of the different operations in Table 7.3 is shown, with the verification being in hardware.

In 6LoWPAN networks, it is often so that the end node does not have a border gateway as a neighbor. In that case, the certificate has to travel through multiple other nodes in order to get through. Naturally, the energy consumption for these intermediate nodes are also a concern. Figure 7.3 shows the total energy consumption for all nodes, including the end node, when a certificate has to travel through the network. Included is also the decoding and verifying that is done by the end node.

What the results show is that the further the certificate has to travel in the network, the bigger gets the difference in energy consumption between compressed and uncompressed certificates. This is somewhat natural, because both certificates follow the curve $y = kx+m$ where y is the energy consumption, k is the consumption for receive and transmit, and m the consumption for decode and verify. When x, the number of hops, grows big, the ratio between compressed and uncompressed converges to the ratio of k.

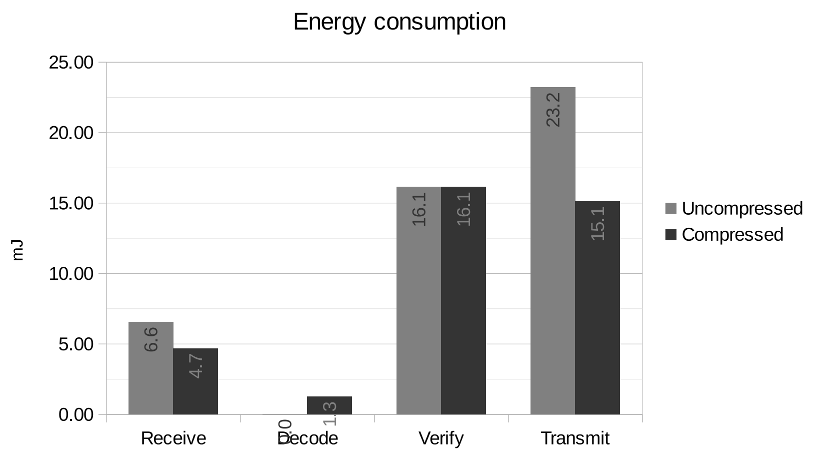

Figure 7.2. Energy consumption for the different parts of the certificate handling process, where hardware cryptography is used.

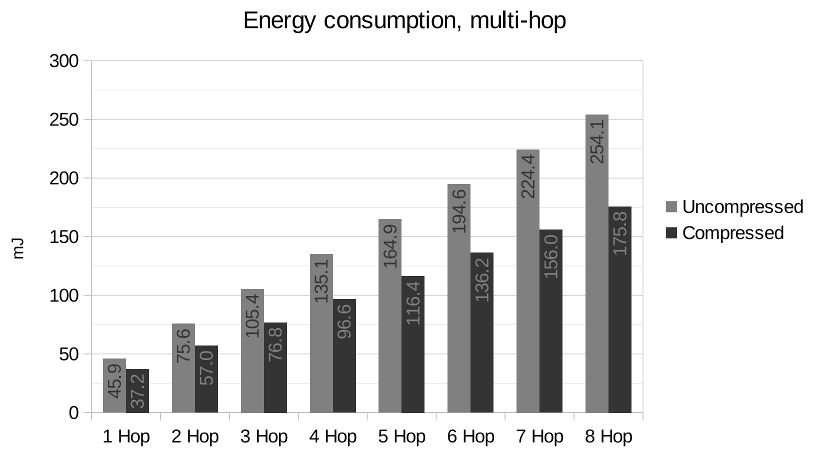

Figure 7.3. Cummulative energy consumption with multiple hops between border gateway and end node for a normal use case of certificate handling, where hardware cryptography is used.

7.3 Compatibility

The output from parsing a profiled certificate in OpenSSL is found in Figure 7.4. The parsing gave no error, which means that the certificate is a valid X.509 certificate. The output also shows that the fields contain values that are compatible with the X.509 Profile for IoT.

```
$ openssl x509 -in xiot_certificate.crt -text -noout

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 42 (0x2a)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = Root CA
        Validity
            Not Before: May 12 08:55:26 2017 GMT
            Not After : May 13 12:42:06 2017 GMT
        Subject: CN = 01-23-45-67-89-AB-CD-EF
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:46:ce:f1:7e:f1:a0:1c:b5:af:31:fe:bd:ca:c3:
                    cc:86:73:83:2b:20:9a:af:2b:65:c1:e6:10:d7:c3:
                    f7:6b:5d:d7:e2:65:d1:3a:95:f7:4d:3e:88:da:51:
                    de:ec:fd:90:24:cf:42:7d:05:40:97:83:86:6b:13:
                    dc:35:42:23:09
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature, Certificate Sign
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:c4:50:ad:01:26:c9:b1:3d:92:f7:09:28:ec:
         3d:0b:59:ee:23:3d:4a:0c:97:39:cf:f4:2b:38:58:99:c7:6b:
         7a:02:21:00:b4:20:ef:ff:97:18:e6:58:1e:bf:5a:d8:85:2c:
         96:5a:61:7f:2c:d7:d8:23:83:a4:90:23:b5:c1:54:dd:e5:f8
```

Figure 7.4. Output from parsing a certificate created by the provided implementation with OpenSSL.

Chapter 8

Conclusions

This is the final chapter that concludes what have been done in this thesis and what is written in this report. It is discussed whether the goals have been met, if the proposed solution is secure or not, and what work is left to do.

8.1 Verdict

The results show that adding the extra complexity by compressing and decompressing certificates does not weigh up the decreased energy usage from less radio transmission. In all cases, it is more energy efficient to use the compressed version and the more hops the certificates travel within the 6LoWPAN network before reaching the end node, the more energy is saved. This means that the goal of a more energy efficient solution is met.

Without compression, the X.509 Profile for IoT helps to keep the certificate size and complexity down. To know the exact extent of the gains from the profile, a more thorough test has to be made.

The certificates created was proven to be X.509 compatible and should be processable by any entity that supports X.509 certificates.

The code size for compressed certificates was slightly larger than for uncompressed, which is expected since the implementation is built upon it. This does not have to be the case, if the implementation for compressed certificates is optimized and standalone, so the need for external libraries expires.

8.2 Security analysis

One might wonder whether this reduction in size compromises the the security of the protocol. However, all the design decisions have been taken with security in mind and should therefore not make the certificates less secure. Some considerations that could compromise security are the following:

- **Cryptographic strength**. The cryptographic strengths of ECC keys and signatures are much higher than RSA by its nature, given the same key size. It means that the amount of computation needed to find a private key from a signature or public key is significantly harder for ECC than for RSA. Therefore the same security level can be reached with smaller keys and signatures. This has been discussed in section 3.7. The security level for this profile has been chosen to be high enough until year 2031 and further.

- **Certificate structure**. The certificates following the X.509 Profile for IoT are still valid X.509 certificates and their structure are therefore as secure as previously. Since keys are represented the same way and signatures are generated the same way, no compromises have been made that could weaken the security.

- **Limited issuer and subject fields**. By reducing the issuer and subject to only be a text string increases the probability of name collisions significantly. For node subjects, this has been resolved by using the EUI-64, which is unique world wide, with the downside of being less human friendly. For the issuer and CA subjects, this does not have to be a problem. If a CA happens to have the same name as another CA, it would still need the other CA's private key in order to cause trouble. The nodes do not verify the issuer by its name only, but with the pre-stored public key in the first hand, so if the CA name is as expected and the signature is valid, it does not matter that another CA might have the same name. The biggest threat is that it can cause confusion among humans.

- **Non mandatory extensions**. This profile have ruled four extensions that previously was mandatory to process as voluntary. This is not a threat in the sense of validating invalid certificates, but more so that the certificate might be used in a way that was not intended. In such a constrained environment it is unlikely that, for example, a sensor node starts to act as a CA. There might however be a case when these extensions are necessary, which will lead to two devices that can't communicate with each other if the constrained node has chosen not to process them.

- **Compromised border gateways**. As with all computers and units, a border gateway can somehow be compromised by an adversary. The adversary can then insert malicious software that incorrectly compresses the certificate. Anyhow, if a certificate gets altered with in any way, it could be in the compressed format or not, the signature of the certificate will not be valid. The nature of certificates implies that certificates can be, and usually is, sent as plain text and are needed in order to set up a secure link in the first place.

- **Denial of Service (DOS) attacks**. Since an extra step, the uncompressing of the tbs structure, is added to the certificate verification step at the end node, this can be exploited to perform a DOS attack on the nodes. By sending large amounts of invalid certificates, an adversary can force the node to decompress the certificates and verify the signatures just to find out that the certificates are invalid, as there is no way of knowing this before verifying them.

8.3 Future work

The work that has been done in this thesis should not be considered as complete. There are several ways to improve the lightness and robustness for the profile and the compression tool.

8.3.1 Hardware

In this thesis, the implementation has only been tested on one specific platform. To prove that this is a generic solution that can be adapted by many different platforms, further tests needs to be done. It could also be worthwhile to determine the lower bound of which this solution can be applied to.

8.3.2 Software

The software implementation of the compression tool given in this thesis is not a fully optimized and minimal library and can be further developed. The software can be optimized for size and complexity and should be tested for correctness. The implementation can be tailored for this specific use case and external libraries could be removed to reduce memory and increase efficiency.

8.3.3 Deeper evaluation

To determine the effect of the X.509 Profile for IoT, a deeper and more thoroughly evaluation has to be made. It might be needed to monitor a system before and after the deployment of profiled certificates. It also needs to be evaluated on a higher level, such as ease of programming and testing of such a system.

8.3.4 Further cryptography efficiency

This profile is designed to not compromise the security and cryptographic strength when deciding on which public key cryptography, signature and hashing algorithm to use. Other solutions that are more lightweight, but less secure, can be sufficient for some systems where an even more lightweight solution is favorable. Such solutions need to be carefully evaluated and the trade-offs need to be thoroughly pronounced.

Bibliography

[1] Stephen E. Deering and Robert M. Hinden. Internet Protocol, Version 6 (IPv6) Specification, May 2017, URL: https://tools.ietf.org/id/draft-ietf-6man-rfc2460bis-13.html (Accessed 2017-06-05).

[2] Nandakishore Kushalnagar, Gabriel Montenegro, David E. Culler, and Jonathan W. Hui. RFC 4944 - transmission of IPv6 Packets over IEEE 802.15.4 Networks, September 2007, URL: https://tools.ietf.org/html/rfc4944 (Accessed 2017-01-24).

[3] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP), March 2013, URL: https://tools.ietf.org/html/draft-ietf-core-coap-14 (Accessed 2017-01-24).

[4] Tim Dierks and Eric Rescorla. RFC 5246 - the Transport Layer Security (TLS) Protocol Version 1.2, August 2008, URL: https://tools.ietf.org/html/rfc5246 (Accessed 2017-01-24).

[5] Eric Rescorla and Nagendra Modadugu. RFC 6347 - Datagram Transport Layer Security Version 1.2, January 2012, URL: https://tools.ietf.org/html/rfc6347 (Accessed 2017-06-06).

[6] Russell Housley et al. RFC 2459 - internet X.509 Public Key Infrastructure Certificate and CRL Profile, January 1999, URL: https://tools.ietf.org/html/rfc2459 (Accessed 2017-01-20).

[7] Dave Cooper. RFC 5280 - internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, May 2008, URL: https://tools.ietf.org/html/rfc5280 (Accessed 2017-01-19).

[8] International Telecommunication Union ITU. Introduction to ASN.1, URL: http://www.itu.int:80/en/ITU-T/asn1/Pages/introduction.aspx (Accessed 2017-02-01).

[9] Carsten Bormann and Paul Hoffman. RFC 7049 - concise Binary Object Representation (CBOR), October 2013, URL: https://tools.ietf.org/html/rfc7049 (Accessed 2017-01-20).

[10] W3Schools. JSON Introduction, URL: http://www.w3schools.com/js/js_json_intro.asp (Accessed 2017-02-01).

[11] Christoph Vigano and Henk Birkholz. CBOR data definition language (CDDL): a notational convention to express CBOR data structures, September 2016, URL: https://tools.ietf.org/html/draft-greevenbosch-appsawg-cbor-cddl-09#appendix-E (Accessed 2017-01-31).

[12] Contiki: The Open Source Operating System for the Internet of Things, URL: http://www.contiki-os.org/index.html (Accessed 2017-02-13).

[13] S. Raza, S. Duquennoy, T. Chung, D. Yazar, T. Voigt, and U. Roedig. Securing communication in 6lowpan with compressed IPsec. In *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, pages 1–8, June 2011.

[14] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt. Lithe: Lightweight Secure CoAP for the Internet of Things. *IEEE Sensors Journal*, 13(10):3711–3720, October 2013. ISSN 1530-437X.

[15] Max Pritikin and David McGrew. The Compressed X.509 Certificate Format, May 2010, URL: https://tools.ietf.org/html/draft-pritikin-comp-x509-00 (Accessed 2017-01-23).

[16] Peter Deutsch. RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3, May 1996, URL: https://tools.ietf.org/html/rfc1951 (Accessed 2017-03-06).

[17] Graham Edgecombe. Compressing X.509 certificates, December 2016, URL: https://www.grahamedgecombe.com/blog/2016/12/22/compressing-x509-certificates (Accessed 2017-03-07).

[18] Hannes Tschofenig and Thomas Fossati. RFC 7925 - Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things, July 2016, URL: https://tools.ietf.org/html/rfc7925 (Accessed 2017-03-20).

[19] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz. Energy analysis of public-key cryptography for wireless sensor networks. In *Third IEEE International Conference on Pervasive Computing and Communications*, pages 324–328, March 2005.

[20] Peter Gutmann. X.509 Style Guide, October 2000, URL: https://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt (Accessed 2017-02-16).

[21] Karen A Lambert. Guidelines for 64-bit Global Identifier (EUI-64), January 2015, URL: https://standards.ieee.org/develop/regauth/tut/eui64.pdf (Accessed 2017-05-04).

[22] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, September 1977, URL: http://people.csail.mit.edu/rivest/Rsapaper.pdf (Accessed 2017-02-02).

[23] Victor S. Miller. Use of Elliptic Curves in Cryptography, 1986, URL: http://link.springer.com/content/pdf/10.1007%2F3-540-39799-X_31.pdf (Accessed 2017-02-03).

[24] Kerry Maletsky. RSA vs ECC Comparison for Embedded Systems, 2015, URL: http://www.atmel.com/Images/Atmel-8951-CryptoAuth-RSA-ECC-Comparison-Embedded-Systems-WhitePaper.pdf (Accessed 2017-02-01).

[25] Elaine Barker. Recommendation for Key Management: General, January 2016, URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (Accessed 2107-02-01).

[26] Tim Polk, Russell Housley, and Larry Bassham. RFC 3279 - Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and CRL Profile, April 2002, URL: https://tools.ietf.org/html/rfc3279 (Accessed 2017-02-20).

[27] Scott A. Vanstone. Compressed ECDSA signatures, November 2007, URL: http://www.google.com/patents/US8631240 (Accessed 2017-04-18).

[28] Sean Turner, Kelvin Yiu, Daniel R. L. Brown, Russ Housley, and Tim Polk. RFC 5480 - Elliptic Curve Cryptography Subject Public Key Information, March 2009, URL: https://tools.ietf.org/html/rfc5480 (Accessed 2017-04-18).

[29] Daniel R. L. Brown. Standards for Efficient Cryptography 1 (SEC 1), May 2009, URL: http://www.secg.org/sec1-v2.pdf (Accessed 2017-05-09).

[30] Texas Instruments. CC2538 Powerful Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6lowpan, and ZigBee© Applications, December 2012, URL: http://www.ti.com/lit/ds/swrs096d/swrs096d.pdf (Accessed 2017-05-24).

[31] Zolertia S.L. Firefly - Zolertia/Resources Wiki, January 2017, URL: https://github.com/Zolertia/Resources/wiki/Firefly (Accessed 2017-05-24).

[32] Adam Dunkels, Fredrik Österlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *EmNets Fourth Workshop on Embedded Networked Sensors*, pages 28–32, 2007.

Appendix A

Example certificates used

These are the certificates used when doing the evaluation and comparisons.

A.1 X.509 Profile for IoT

Base64 encoded:

```
-----BEGIN CERTIFICATE-----
MIIBQDCB5qADAgECAgEqMAoGCCqGSM49BAMCMBIxEDAOBgNVBAMMB1Jvb3QgQ0Ew
HhcNMTcwNTEyMDg1NTI2WhcNMTcwNTEzMTI0MjA2WjAiMSAwHgYDVQQDDBcwMS0y
My00NS02Ny04OS1BQi1DRC1FRjBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABEbO
8X7xoBy1rzH+vcrDzIZzgysgmq8rZcHmENfD92td1+Jl0TqV900+iNpR3uz9kCTP
Qn0FQJeDhmsT3DVCIwmjHTAbMAwGA1UdEwEB/wQCMAAwCwYDVR0PBAQDAgKEMAoG
CCqGSM49BAMCA0kAMEYCIQDEUK0BJsmxPZL3CSjsPQtZ7iM9SgyXOc/0KzhYmcdr
egIhALQg7/+XGOZYHr9a2IUsllphfyzX2CODpJAjtcFU3eX4
-----END CERTIFICATE-----
```

A.2 Compressed

Hexadeximal values:

```
87 18 2a 67 52 6f 6f 74 20 43 41 82 1a 59 15 78
7e 1a 59 16 ff 1e 48 01 23 45 67 89 ab cd ef 58
21 03 46 ce f1 7e f1 a0 1c b5 af 31 fe bd ca c3
cc 86 73 83 2b 20 9a af 2b 65 c1 e6 10 d7 c3 f7
6b 5d 82 83 13 f5 42 30 00 82 0f 44 03 02 02 84
58 40 c4 50 ad 01 26 c9 b1 3d 92 f7 09 28 ec 3d
0b 59 ee 23 3d 4a 0c 97 39 cf f4 2b 38 58 99 c7
6b 7a b4 20 ef ff 97 18 e6 58 1e bf 5a d8 85 2c
96 5a 61 7f 2c d7 d8 23 83 a4 90 23 b5 c1 54 dd
e5 f8
```

A.3 Regular X.509

Base64 encoded:

```
-----BEGIN CERTIFICATE-----
MIIF8jCCBNqgAwIBAgIQDmTF+8I2reFLFyrrQceMsDANBgkqhkiG9w0BAQsFADBw
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMS8wLQYDVQQDEyZEaWdpQ2VydCBTSEEyIEhpZ2ggQXNz
dXJhbmNlIFNlcnZlciBDQTAeFw0xNTExMDMwMDAwMDBaFw0xODExMjgxMjAwMDBa
MIGlMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEUMBIGA1UEBxML
TG9zIEFuZ2VsZXMxPDA6BgNVBAoTM0ludGVybmV0IENvcnBvcmF0aW9uIGZvciBB
c3NpZ25lZCBOYW1lcyBhbmQgTnVtYmVyczETMBEGA1UECxMKVGVjaG5vbG9neTEY
MBYGA1UEAxMPd3d3LmV4YW1wbGUub3JnMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCgKCAQEAs0CWL2FjPiXBl61lRfvvE0KzLJmG9LWAC3bcBjgsH6NiVVo2dt6u
Xfzi5bTm7F3K7srfUBYkLO78mraM9qizrHoIeyofrV/n+pZZJauQsPjCPxMEJnRo
D8Z4KpWKX0LyDu1SputoI4nlQ/htEhtiQnuoBfNZxF7WxcxGwEsZuS1KcXIkHl5V
RJOreKFHTaXcB1qcZ/QRaBIv0yhxvK1yBTwWddT4cli6GfHcCe3xGMaSL328Fgs3
jYrvG29PueB6VJi/tbbPu6qTfwp/H1brqdjh29U52Bhb0fJkM9DWxCP/Cattcc7a
z8EXnCO+LK8vkhw/kAiJWPKx4RBvgy73nwIDAQABo4ICUDCCAkwwHwYDVR0jBBgw
FoAUUWj/kK8CB3U8zNllZGKiErhZcjswHQYDVR0OBBYEFKZPYB4fLdHn8SOgKpUW
5Oia6m5IMIGBBgNVHREEejB4gg93d3cuZXhhbXBsZS5vcmeCC2V4YW1wbGUuY29t
ggtleGFtcGxlLmVkdYILZXhhbXBsZS5uZXSCC2V4YW1wbGUub3Jngg93d3cuZXhh
bXBsZS5jb22CD3d3dy5leGFtcGxlLmVkdYIPd3d3LmV4YW1wbGUubmV0MA4GA1Ud
DwEB/wQEAwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwdQYDVR0f
BG4wbDA0oDKgMIYuaHR0cDovL2NybDMuZGlnaWNlcnQuY29tL3NoYTItaGEtc2Vy
dmVyLWc0LmNybDA0oDKgMIYuaHR0cDovL2NybDQuZGlnaWNlcnQuY29tL3NoYTIt
aGEtc2VydmVyLWc0LmNybDBMBgNVHSAERTBDMDcGCWCGSAGG/WwBATAqMCgGCCsG
AQUFBwIBFhxodHRwczovL3d3dy5kaWdpY2VydC5jb20vQ1BTMAgGBmeBDAECAjCB
gwYIKwYBBQUHAQEEdzB1MCQGCCsGAQUFBzABhhhodHRwOi8vb2NzcC5kaWdpY2Vy
dC5jb20wTQYIKwYBBQUHMAKGQWh0dHA6Ly9jYWNlcnRzLmRpZ2ljZXJ0LmNvbS9E
aWdpQ2VydFNIQTJIaWdoQXNzdXJhbmNlU2VydmVyQ0EuY3J0MAwGA1UdEwEB/wQC
MAAwDQYJKoZIhvcNAQELBQADggEBAISomhGn2L0LJn5SJHuyVZ3qMIlRCIdvqe0Q
6ls+C8ctRwRO3UU3x8q8OH+2ahxlQmpzdC5al4XQzJLiLjiJ2Q1p+hub8MFiMmVP
PZjb2tZm2ipWVuMRM+zgpRVM6nVJ9F3vFfUSHOb4/JsEIUvPY+d8/Krc+kPQwLvy
ieqRbcuFjmqfyPmUv1U9QoI4TQikpw7TZU0zYZANP4C/gj4Ry48/znmUaRvy2kvI
l7gRQ21qJTK5suoiYoYNo3J9T+pXPGU7Lydz/HwW+w0DpArtAaukI8aNX4ohFUKS
wDSiIIWIWJiJGbEeIO0TIFwEVWT0nbNl/faPXpk5IRXicapqiII=
-----END CERTIFICATE-----
```

Appendix B

Certificate specification

B.1 Compressed (CBOR CDDL)

```
certificate = [
    serial_number   : uint,
    issuer          : text,
    validity        : [notBefore: int, notAfter: int],
    subject         : text / bytes,
    public_key      : bytes,
    ? extensions    : [+ extension],
    signature       : bytes
]

extension = [
    oid             : int,
    ? critical      : bool,
    value           : bytes
]
```

B.2 X.509 Profile (ASN.1)

```
XIOTCertificate DEFINITIONS EXPLICIT TAGS ::= BEGIN

Certificate          ::=   SEQUENCE {
   tbsCertificate            TBSCertificate,
   signatureAlgorithm        SignatureIdentifier,
   signature                 BIT STRING
}

TBSCertificate       ::=   SEQUENCE {
   version              [0]  INTEGER {v3(2)},
   serialNumber              INTEGER (1..MAX),
   signature                 SignatureIdentifier,
   issuer                    Name,
   validity                  Validity,
   subject                   Name,
   subjectPublicKeyInfo      SubjectPublicKeyInfo,
   extensions           [3]  Extensions OPTIONAL
}

SignatureIdentifier  ::=   SEQUENCE {
   algorithm                 OBJECT IDENTIFIER (ecdsa-with-SHA256)
}

Name                 ::=   SEQUENCE SIZE (1) OF DistinguishedName
DistinguishedName    ::=   SET SIZE (1) OF CommonName
CommonName           ::=   SEQUENCE {
   type                      OBJECT IDENTIFIER (id-at-commonName),
   value                     UTF8String
              -- If CA, value is CA name, else EUI-64 in format
              -- "01-23-54-67-89-AB-CD-EF"
}

Validity             ::=   SEQUENCE {
   notBefore                 UTCTime,
   notAfter                  UTCTime
              -- In format "YYMMDDhhmmssZ"
}

SubjectPublicKeyInfo ::=   SEQUENCE {
   algorithm                 AlgorithmIdentifier,
   subjectPublicKey          BIT STRING
}
```

```
AlgorithmIdentifier  ::=   SEQUENCE {
   algorithm                 OBJECT IDENTIFIER (id-ecPublicKey),
   parameters                OBJECT IDENTIFIER (prime256v1)
}

Extensions           ::=   SEQUENCE SIZE (1..MAX) OF Extension

Extension            ::=   SEQUENCE {
   extnId                    OBJECT IDENTIFIER,
   critical                  BOOLEAN DEFAULT FALSE,
   extnValue                 OCTET STRING
}

ansi-X9-62           OBJECT IDENTIFIER   ::=
                       {iso(1) member-body(2) us(840) 10045}

id-ecPublicKey       OBJECT IDENTIFIER   ::=
                       {ansi-X9-62 keyType(2) 1}

prime256v1           OBJECT IDENTIFIER   ::=
                       {ansi-X9-62 curves(3) prime(1) 7}

ecdsa-with-SHA256    OBJECT IDENTIFIER   ::=
                       {ansi-X9-62 signatures(4) ecdsa-with-SHA2(3) 2}

id-at-commonName     OBJECT IDENTIFIER   ::=
                       {joint-iso-itu-t(2) ds(5) attributeType(4)  3}

END
```

TRITA EE 2017:085