Implementation and Evaluation of Datagram Transport Layer Security (DTLS) for the Android Operating System

DANIELE TRABALZA

KTH Electrical Engineering

Master's Degree Project
Stockholm, Sweden June 2012

Abstract

Smartphones are nowadays a tool that everyone posses. With the replacement of the IPv4 with the IPv6 it is possible to connect to the Internet an extremely large number of electronic devices. Those two factors are the premises to use smartphones to access those devices over a hybrid network, composed of Wireless Sensor Networks, IPv6-based Internet of Things, constrained networks and the conventional Internet. Some of these networks are very lossy and use the UDP protocol, hence the most suitable protocol to access resources is CoAP, a connection-less variant of the HTTP protocol, standardized as web protocol for the Internet of Things. The sensitivity of information and the Machine-to-Machine interaction as well as the presence of humans make the End-to-End security one of the requirements of the IPv6 Internet of Things. Secure CoAP (CoAPS) provide security for the CoAP protocol in this context.

In this thesis secure CoAP for Android smartphones is designed implemented and evaluated, which is at the moment the first work that enables CoAPS for smartphones. All the cryptographic cipher suites proposed in the CoAP protocol, among which the pre-shared key and certificate-based authentications are implemented, using the Elliptic Curve Cryptography and the AES algorithm in the CCM mode.

The feasibility of this implementation is evaluated on a Nexus phone, which takes the handshake time in order to exchange parameters to secure the connection to about five seconds, and an increase from one to three seconds of the DTLS retransmission timer. A part for this initial delays the performances using secure CoAP are comparable to the performances obtained using the same protocol without security. The implementation allows also to secure the UDP transport thanks to the DTLS implementation, allowing any potential application to exchange secure data and have mutual authentication.

List of acronyms and abbreviations

IoT	Internet of Things
TCP	Transmission Control Protocol
TLS	Transport Later Security
CoAP	Constrained Application Protocol
DTLS	Datagram Transport Layer Security
DH	Diffie-Hellman
ECC	Elliptic Curve Cryptography
6LoWPAN	Low power Wireless Personal Area Networks
OSI	Open Systems Interconnection model
IPSec	Internet Protocol Security
UDP	User Datagram Protocol
PKI	Public Key Infrastructure
MTU	Maximum Transmission Unit
SSL	Secure Socket Layer
HTTP	Hypertext Transfer Protocol
FTP	File Transfer Protocol
SMTP	Simple Mail Transport Protocol
MTU	Maximum Transmission Unit
IMAP	Internet Message Access Protocol
POP	Post Office Protocol
SIP	Session Initiation Protocol
DoS	Denial Of Service
M2M	Machine To Machine
CoAPS	Constrained Application Protocol Secure
JNI	Java Native Interface
ICS	Ice Cream Sandwich
IETF	Internet Engineering Task Force
AEAD	Authenticated Encryption with Associated Data
MAC	Message Authentication Code

MITM	Man In The Middle
PKI	Public Key Infrastructure
DSA	Digital Signature Algorithm
WSN	Wireless Sensor Network
PRF	Pseudo Random Function
CA	Certification Authority
ECDSA	Elliptic Curve Digital Signature Algorithm
DN	Distinguished Name

List of Figures

Contents

Chapter 1

Introduction

The replacement of the IPv4 with the IPv6 allows the connection of 2^{128} devices to the Internet [5]. It is possible to assign $6.67 * 10^{27}$ IP addresses per square meters, which allows the connection of a large number of human-made object to the Internet, creating a network called the IoT [15] which before the IPv6 was not possible. The IoT requires fast delivery, real time communications and it is often based on multi hope communications. The protocol used to exchange data between sensors is CoAP, in contrast with computer network, which use the Transmission Control Protocol (TCP) protocol and do not have the same requirements. The TCP protocol is secured with mechanisms such as Transport Later Security (TLS) [6] which provide data confidentiality and integrity. Those mechanisms can be high computationally expensive and not suitable for some environments. In the IoT the main protocol, finding a strict correspondence with Hypertext Transfer Protocol (HTTP) is CoAP, which is based on datagrams and it is particularly suitable for constrained and lossy environments [14].l, complicating the implementation.

1.1 Problem Definition

Any CoAP-enabled device can in this way interact with other nodes of the IoT. For instance, a Any CoAP-enabled smartphone can communicate with sensor networks, actuators and so on over the Internet. Some applications might require data confidentiality, which is not possible to provide using common mechanisms used for instance on a similar protocol (HTTP).

Enabling CoAP for smartphones, which is an important and necessary tool in everyday life, will give the possibility to interact with any CoAP-enabled devices for countless purposes: from smart house, to gaming, to health monitoring, and so on.

The purpose of this thesis is hence to allow this possibility, focusing on the security issue that derives from this. It is important to secure data, which can be transmitted over the air, or through the Internet, hence susceptible to attacks. The CoAP RFC [14] suggests possible ways to reach this purpose, which will be later on analysed and evaluated.

1.2 Goal

The objective of this thesis is to enable secure CoAP allowing a smartphone with an Android OS to became a "thing" and interact with the other devices, with an emphasis on data confidentiality and mutual authentication, which is important since the communications are machine to machine with an application that supports both CoAP and CoAPS protocols. Since the Datagram Transport Layer Security (DTLS) protocol use to secure CoAP messages, resides in the application layer, it is possible to address the single application present on the node, giving an automatic key management thanks to the Public Key Infrastructure (PKI).

The process will involve the key exchange algorithm Diffie-Hellman (DH) and the Elliptic Curve Cryptography (ECC), illustrated in section **??** page 14. The implementation will be also able to communicate with other standardized DTLS implementations, which hosts are in a Low power Wireless Personal Area Networks (6LoWPAN) or a conventional network using the the IPv6 protocol.

The work will be then evaluated in order to identify the feasibility of using the CoAP protocol with the newly implemented security mechanisms, highliting:

- the round trip time of requests and responses
- the additional CPU time overhead

1.3 Research Method

With the replacement of the IPv4 with the IPv6 in the market the CoAP protocol became more and more used. As far as security standard is concerned, there is no research material available yet, and most probably this document will be the first one to research the security issues in the CoAP protocol. In the CoAP RFC [14] it is possible to find a set of specification to address this problem, so in this document will be explored, analysing from the bottom the CoAP security and identify future research areas. For this purpose, in the beginning will be used a quantitative approach that is inductive in nature and helps us answer questions such as is it feasible to apply secure CoAP to smartphones and will it be feasible to use it in nowadays smartphones? The use of this approach will help to identify shortcomings of CoAPS and successively perform a CoAPS evaluation. After analysing CoAPS security using the quantitative approach it will be used a qualitative approach, that is deductive in nature as it helps to answer questions such as why is it important to secure CoAP, how it is possible to secure it and what should be done in order to provide a comprehensive solution to secure CoAP communications? The security mechanisms in the CoAP RFC are spread in several documents, and not only there is not a single document explaining security issues comprehensively, but there is neither a full implementation of these specifications. After studying the whole standard and internal mechanisms it will be critically analysed the shortcoming.

1.4 Assumptions

The whole document is based on the IPv6 protocol that is slowly replacing IPv4, hence any reference on network, IP address, and related items are to be consid-

ered based on IPv6 protocol, unless explicitly mentioned. The TLS protocol in the document is referring to the TLS 1.2 [6] unless explicitly mentioned. The DTLS protocol is referring to the version 1.2 [13] that is based on TLS 1.2. The CoAP protocol does not have a standard RFC; the last version is the draft number 12 [14]. The protocol CoAPS does not have a IANA port number assigned yet, so it has given the port number 5684. In the TLS protocol, section 2.2 5, it is assumed that TLS is carried on top of TCP for simplicity. Other reliable protocols in alternative to TCP can be used without altering its functioning.

The whole project is based on the Android OS version 4 (API 14) also known as Ice Cream Sandwich (ICS) since it includes important features for the thesis' purpose such as a unified user interface to install trusted certificates. This will make the application smoother and easier to use, allowing the possibility to install certificates directly from the smartphone, without the need of an additional computer.

1.5 Structure of the thesis

This document starts with an introduction of the problem addressed. It defines the problem and goal, research method and assumptions used in the document.

The next section aims to explain key concepts used in this document, in order to provide a better understanding for the following sections, illustrating the relevant aspects of CoAP, TLS and the DTLS protocols, ending with a deeper explanation of mechanisms and concepts used during the implementation.

Section 3 analyses more in depth the problem illustrated in section 1 showing the currently available options that can be used to reach the goal. The chapter ends with the analysis of pros and cons of the chosen option, and the division of the work in phases used to reach the goal with the chosen option.

Once it has been decided the way to proceed, further analysis and the actual work of applying the preferred solution are explained in section 4.

Section 5 is the evaluation of the work carried on in the previous chapter. Several aspects of the implementation are evaluated, in relation to the existing environment, explaining the reasoning behind the selection of such tests and the accuracy of the results.

The last two sections are the conclusion, showing the result and benefit that can be gained with this work, and possible future work in order to improve the current status or enhance the functionalities.

Chapter 2

Technical Background

This section provides the knowledge needed for a proper understanding of the technologies mentioned in this document. For a better understanding the TLS protocol is initially presented, due to its similarities with DTLS. A detailed explanation of DTLS will follow, pointing out the main differences with TLS. The section will be concluded by an explanation of the cipher suites used during the handshake, encryption techniques and key exchange, illustrating first the general concept and then investigating details that will be reused in the rest of the document.

2.1 CoAP - Constrained Application Protocol

CoAP is a draft protocol that aims to realise a subset of HTTP functions optimised for Machine To Machine (M2M) applications in constrained nodes (i.e. 8bit micro-controllers with limited RAM and ROM) and networks 6LoWPAN [14]. As previously mentioned the CoAP protocol is very similar to the HTTP protocol. The main differences are that usually in M2M environment the end points are often acting both as client and server. Moreover CoAP is based on a datagram protocol, usually User Datagram Protocol (UDP) and the messages are asynchronous, as opposed to the HTTP protocol where synchronous messages are sent.

The optional reliability in the CoAP protocol is obtained by a message type called *Confirmable*, in which the recipient sends an acknowledge to the sender. In the header is present also a a message ID used to detect duplicates. It is also possible to send unreliable messages called *Non-Confirmable*, that are not acknowledged by an *Acknowledgment* message. The last message is the *Reset* message and it is used to specify that a confirmable message was received but the recipient might have been lost its previous state (due to error or rebooting). These fields are specified in the CoAP header.

This protocol specifies also that the message should fit in a single datagram, in order to avoid unnecessary fragmentation.

CoAP protocol uses methods like GET, POST, PUT and DELETE similarly to HTTP, in a Request/Response model. Both requests and responses can be confirmable or non-confirmable but the response might be sent separately from the acknowledge message. Since messages can be asynchronous, the client might

receive the acknowledge that the server received the request, before receiving the actual request. In this case the client will generate a *Confirmable* request and send it to the server. The server will then send an acknowledgment to the client, and when the server computed or retrieved the response, it will be sent to the client. The acknowledge and the response can also be contained in the same datagram.

2.2 TLS - Transport Layer Security

TLS is a widely deployed cryptographic protocol based on Secure Socket Layer (SSL) used to secure traffic for HTTP, Internet Message Access Protocol (IMAP), Post Office Protocol (POP) and other protocols. It belongs to the application layer of the Open Systems Interconnection model (OSI) model and encapsulates protocols such as HTTP, File Transfer Protocol (FTP) and Simple Mail Transport Protocol (SMTP), providing confidentiality and integrity. It uses asymmetric cryptography for key exchange and symmetric cryptography for data encryption and integrity.

TLS provides communication privacy and integrity between two applications, server authentication and optionally client authentication thanks to the Public Key Infrastructure. The protocol is composed of a record layer as carrier and other protocols such as handshake, change cipher spec or alert, that are record layer's payload. TLS requires a reliable protocol on a lower lower level, typically TCP that takes care for retransmissions, packet reordering and message reliability. Transport Later Security allows the generation of session keys to establish a secure communication between two applications and session resumption to optimise resources. After the handshake the communication will be encrypted and integrity protected with the session key generated during the handshake and kept protected thanks to certificates. Typically the client is not authenticated; its authentication is obtained with a username/password pair, while the server is authenticated via its certificate to avoid burden for the clients. After receiving the certificate the client can check its validity and reply back with a session key encrypted with the server's public key, to protect the following messages with a less computational expensive symmetric key. After, client and server can finish the handshake and start the data transmission with the key previously derived.

2.2.1 TLS Full handshake

In this section a TLS full handshake will be explained, in which the client establishes a connection with the server exchanging the parameters needed to protect the data once the handshake is finished. The purpose of the handshake is to exchange parameters to form the same shared secret between the two peers and authenticate the server (and optionally the client). From the shared secret (called pre-master-secret) is then derived the master secret and session keys, depending on the cipher suite used. For simplicity only what is relevant for the comparison with DTLS will be highlighted. Further information is available in [6].

The server is authenticated with its certificate, sent in the handshake message. The client is optionally authenticated. In case the client needs to be

authenticated, in addition to the server *Certificate* message, the server sends a *CertificateRequest* message to the client. When the client receives this message, it will reply with a client *Certificate* and *CertificateVerify*. In this way the peers have mutual authentication.

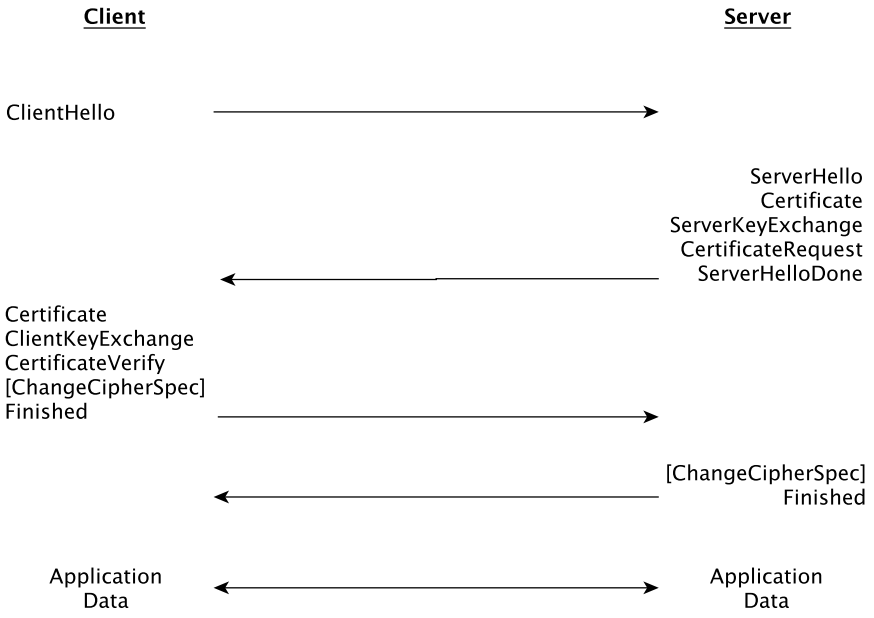

Figure 2.1: TLS Handshake Messages

In figure 2.1 the TLS handshake is shown. The client sends the *ClientHello* message to initialize the handshake. If it contains an existing *sessionID* verified by the server, an abbreviated handshake will be performed, otherwise the full handshake will be continued. This message contains also a list of possible cipher suites and compression methods to be used, as well as a random number used to generate a session key.

The server replies with a *ServerHello*, that includes a session ID (that may be used later for session resuming), the second random number and chosen cipher suite and compression method among the proposed ones in the *ClientHello* message. The server will send then a *Certificate* message, including its certificate, and the *ServerHelloDone* message to indicate the end of the server hello messages, since there can be up to five messages in this phase.

The client has now all the information to calculate the session key that will be used, so it will generate and encrypt it using the server's public key contained in the certificate, and send a *ClientKeyExchange* message. The length and format of the key depends on the previously negotiated algorithm. After this message the client will send a *ChangeCipherSpec* message indicating that next messages will be sent using the previously negotiated keys for integrity and encryption. In figure 2.1 the *ChangeCipherSpec* message is shown in square bracket because it is technically part of the handshake, but this message is not contained in the handshake protocol. The last message sent by the client is the encrypted

Finished messages indicating the end of the handshake.

With the *ChangeCipherSpec* and *Finished* messages the server completes the handshake. After the last message is received it is possible to exchange encrypted application data using the session-based symmetric keys previously negotiated. Depending on the cipher suites negotiated, it might be used one key for both integrity and privacy. This is analysed and explained in section 2.5 page 13.

2.2.2 TLS Record Layer

As shown in figure 2.2, a TLS record layer is composed of content type, to specify how to interpret the payload (fragment), protocol version, and length. Content type can be one of the upper level protocols such as Handshake, Change Cipher Spec, Alert or Application Data. The handshake record carries the handshake messages between the client and the server; the change cipher spec is used during the handshake even if it is not part of the handshake protocol, and it notifies that the next messages are encrypted with the previously negotiated parameters. Alert messages represent error messages happening at protocol level and the last possible payload for the record protocol is the actual application data.

Figure 2.2: TLS Record Message Format

The size of the fields shown are fixed for content type, version and length. The size of fragment is variable and it depends on the upper protocols and the Maximum Transmission Unit (MTU) since the record layer might be fragmented.

Each record is compressed and whenever possible encrypted (the communication is not encrypted in the first messages of the handshake). Since the record layer lies on top of TCP, it offers an implicit sequence number used by TLS in the session keys to introduce liveness.

Figure 2.3: TLS Handshake Message Format

Figure 2.3 shows the handshake protocol. One can notice that the length contained in this protocol is bigger than the length of the record layer, even though the handshake is encapsulated in the record layer. This is because a handshake message size might exceed the size of the MTU allowed by the network (i.e. when sending a certificate) and hence be fragmented. This protocol

carries messages such as Client Hello, Server Hello and Finished, further explained in the next section.

2.3 DTLS - Datagram Transport Layer Security

Under certain circumstances, protocols used for real time applications like Session Initiation Protocol (SIP), require the use of UDP instead of TCP for several reasons: fast delivery is preferred to reliable delivery, and packet loss is acceptable in order to avoid delays deriving from packet reordering or retransmission. In this context it is not possible to use TLS because it requires a reliable protocol that is, for its nature, slower than a datagram based protocol.

For some applications Internet Protocol Security (IPSec) is also not practical to use, mainly because it is placed in a lower level and it cannot serve a single application, but also because it requires higher effort for the design. DTLS solves these problems adapting the TLS protocol for functioning in datagram communications by introducing explicit counters and messages.

2.3.1 DTLS Full handshake

The DTLS full handshake will be now explained. As shown in picture 2.4, the DTLS handshake is very similar to the TLS handshake (picture 2.1).

Due to the unreliability of the datagram protocol, DTLS incorporates mechanisms to retransmit the packet. For instance, if the client sends a *ClientHello*, it expects to receive a *HelloverifyRequest* message from the server. If this doesn't happen, it means that either the *ClientHello* or the *HelloVerifyRequest* is lost, hence after a certain amount of time the client will resend the *ClientHello*. The suggested timer value is between 500 and 1000ms, and good implementations should back off their retransmission timers [12].

The purpose of the *HelloVerifyRequest* is though not related with the retransmission mechanism that both client and server integrate. Its purpose is to avoid Denial Of Service (DoS) attacks that in UDP are very easy to be performed through the exchange of a cookie through exchanging a cookie. A cookie is a value generated by the server based on client's parameter.

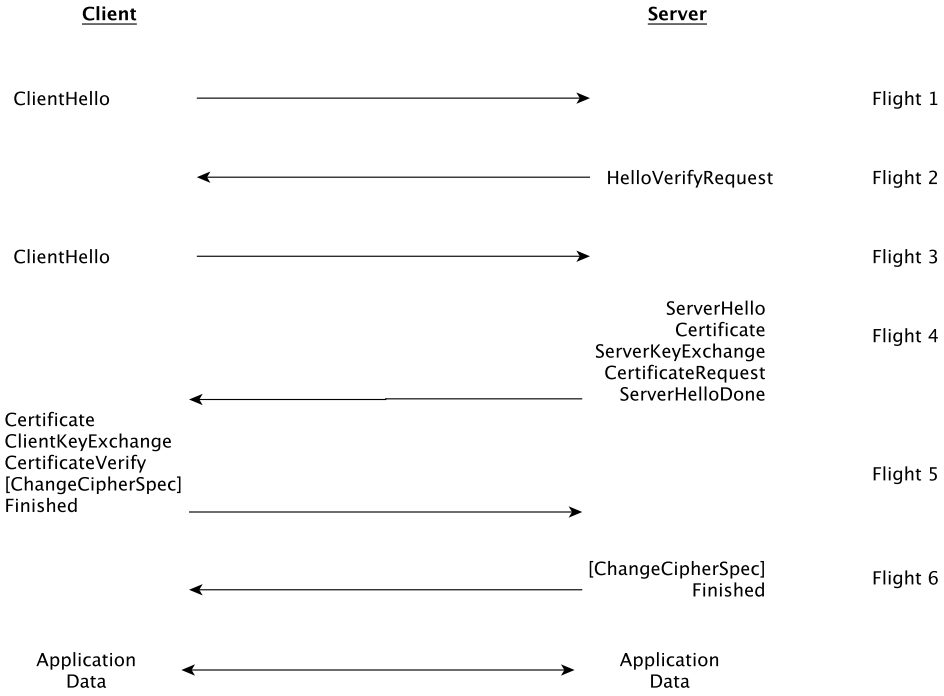

Figure 2.4: DTLS Handshake Messages

The final part of the handshake might seem similar to TLS but since the transport is not reliable, the last message could be lost. If, for example, the last flight is lost, the server believes that the handshake has been completed. Since the client is waiting for the sever *Finished* message, it will retransmit the last flight completing the handshake. This situation does not happen with TLS since there is no packet loss (or in other words, the TCP transport handles retransmissions to avoid packet loss) and needs to be handled with a retransmission timer in DTLS.

The DTLS server might optionally send a *HelloRequest* to ask the client a renegotiation in the same way as it happens with TLS.

2.3.2 DTLS Record Layer

As TLS, the DTLS protocol is composed of several protocols: the basic protocol is called *Record Protocol* and it is shown in figure 2.5. The record layer encapsulates other protocols, such as the *Handshake*, *Alert*, *ChangeCipherSpec* and *Application data*.

It is possible to notice that the DTLS (figure 2.5) and TLS record protocol (figure 2.2 page 7) differ only on two field introduced by DTLS. Those are *epoch* and *sequence_number*. The latter is a sequence number incremented every transmission or retransmission to determine message order and detect duplications. The *epoch* is incremented after each *ChangeCipherSpec* message is sent. These two fields, the 16bit *epoch* and the 48bit *sequence_number* together form the

equivalent of the implicit 64bit sequence number used in TLS. Since in TLS the transport is reliable, it is not needed to carry these counters with the messages. Since no message is lost both peers will have always the same counter value. In case of DTLS instead, where packet might be lost, those values are carried explicitly in each record.

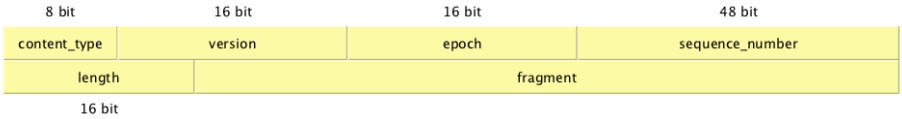

Figure 2.5: DTLS Record Message Format

As in the previous case, the TLS handshake protocol (figure 2.3 page 7) and the DTLS handshake protocol (figure 2.6) differ from each other by the fields *message_sequence*, *fragment_offset* and *fragment_length*.

Message_sequence indicates the message number for both client and server. Both start from zero and increment this field every time a new handshake message is sent. This field is however not incremented with retransmission as it happens with the sequence_number since it identifies the exact state of the handshake which is independent of retransmissions.

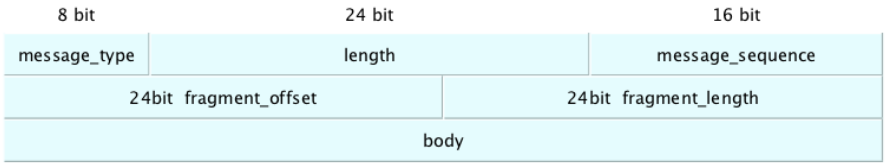

Figure 2.6: DTLS Handshake Message Format

An Handshake message can be bigger than the maximum size allowed by the network to a datagram (MTU). For this purpose the DTLS protocol offers a mechanism to fragment the message over more records and recompose it in the recipient. This is obtained with the fields *fragment_length*, that specifies the number of bytes contained in the current message, and the *fragment_offset*, that specifies the number of bytes contained in previous fragments. If these fields are not overlapping, with the total length that is the same for all the fragments, it is possible to reconstruct the original message.

2.3.3 Example of a DTLS ClientHello message

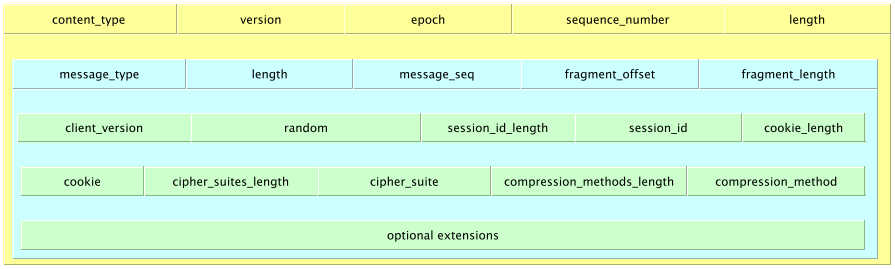

Figure 2.7: DTLS Client Hello Message

Figure 2.7 shows an example of a DTLS *ClientHello* message. With the colour yellow is identified the record layer which differently from the TLS protocol contains the two extra fields epoch and sequence number.

With the cyan is highlighted the Fragment that contains only *Handshake* messages in order to handle datagram fragmentation, reconstructed thanks to the fields message_seq, fragment_offset and fragment_length. Other protocols such as *ChangeCipherSpec* and *Alert* are contained directly in the record's payload.

The green colour identifies the Handshake messages, containing different fields depending on the type of handshake message (figure 2.4). In the hello messages it is possible to find an optional field for extensions. This field is used with elliptic curves to handshake additional parameters.

2.4 Confidentiality and Authenticity

One of the ways to provide data confidentiality is encryption: a way to make data readable only to whom is in possession of certain knowledge, called key. If the key is kept secret, then only who posses it should be able to decipher data (this process is called decryption). The encryption / decryption process transforms the so called plain text (plain data) into cipher text (encrypted data) and vice versa, using a certain algorithm (called cipher), a certain key and other parameters. There are two main classes of algorithm of encryption/decryption: symmetric key and asymmetric key.

A symmetric key algorithm uses a key for encryption, transforming the plain text in cipher text and then the same key is used to reverse the process, obtaining back the plain text given the cipher text and the encryption key. This is the simplest form of encryption and even though it adds operations before sending and receiving data, it is computationally easy to perform these operations.

An asymmetric key algorithm uses on the other hand, two different keys for encryption and decryption, called key pair. A key pair is composed of two mathematically linked keys, called public key and private key. The peculiarity of this key pair is that what is encrypted with the private key, can be decrypted only with the public key that pairs with the one used for encryption. Applying the process in the other way, it is possible to decrypt with the private key, only

what has been encrypted with the mathematically linked public key. This offers not only the possibility to encrypt but also the possibility to use the private key to sign certain data, that can be then validated using the pairing public key. After a successful validation the sender is verified, avoiding attacks such as Man In The Middle (MITM).

Those two types of encryption are often used together for several reasons. The asymmetric encryption, despite the benefits illustrated above, requires a high computational power in order to be performed. It is then not convenient to use it to encrypt every single data. On the other hand, with the symmetric encryption, that operates in less time and can be obtained at hardware level and optimised, it is not possible to have authentication, divulge the public key to verify the origin, and so on. So, taking the best from both, handshake algorithms use a combination of the two: the asymmetric key is used to authenticate the peers and exchange the secret key, that will be then used for the real data encryption / decryption.

2.5 AEAD Cipher Suites

Usually cryptographic applications do not only require to have the data encrypted, hence confidential, but also the data to be available only to authorised people. The reason behind this is that an attacker can intercept this information, and change it, replacing part or all with other information encrypted in the same way so the recipient will not understand the modification. In order to solve this problem, a Message Authentication Code (MAC) is used, which provides the integrity of a message, and this is used often in combination with encryption to provide both confidentiality and integrity.

Recently the idea of using the same algorithm to provide both confidentiality and integrity has become accepted. In opposition with previous algorithms, that require the generation of the MAC to preserve integrity and then encrypt it with the plain text to add confidentiality, this new algorithm in one process gives as result directly an authenticated encryption. This interface is described in the RFC 5116 [9] and in very high level in this section since it is the interface used from one of the implemented cipher suites. Authenticated Encryption with Associated Data (AEAD) allows optimisations to reduce the amount of computations in order to perform encryption and decryption and storage requirements by providing a uniform interface to access cryptographic services. This is needed since the cipher suite on which this document focuses uses the AEAD parameters.

The AEAD interface is composed of two operations: an authenticated encryption and an authenticated decryption.

Authenticated Encryption In order to perform encryption with the possibility to verify its integrity, there are four input parameters needed:

- The Secret Key (K) used to encrypt the plain text
- A Nonce (N) unique within the same key (unless zero-length nonce)
- The Associated Data (A) that will be authenticated but not encrypted
- A Plaintext P which is the data to be encrypted and authenticated

These parameters will produce a cipher text C containing the encrypted data and a way to ensure its integrity.

Authenticated Decryption The authenticated decryption has as input parameters:

- The Secret Key (K) used to encrypt the plain text
- A Nonce (N) unique within the same key (unless zero-length nonce)
- The Associated Data (A) that will be authenticated but not encrypted
- A Cipher text (C) which is the data to be decrypted and authenticated

Whenever the result of the authenticated decryption is a plaintext (P) the integrity of the associated parameters and of the plaintext/cipher text is assured (assuming the AEAD algorithm is secure). If the decryption fails this means that one or more parameters cannot be authenticated.

2.6 Elliptic curve cryptography

Elliptic curves are mathematical functions defined by the equation:
$y^2 = x^3 + a * x + b$,
where x, y, a, and b are elements of the field F_p, and the discriminant is nonzero [11]. These curves holds an important property that can be used in cryptography: they allow, as the discrete logarithm problem used in asymmetric cryptography, to generate two mathematically connected points having certain properties, in which given one point it is infeasible to find the correspondent one. This is used in order to create the key pair used in asymmetric cryptography (illustrated in the next section) that is more efficient than the discrete logarithm problem. It is possible in fact to obtain the same strength with less bytes composing the keys, thus overhead.

This is applied in both authentication and key exchange [3] purposes, as explained in the following sections.

2.7 Asymmetric Encryption

Asymmetric encryption is nowadays a well known concept, on which most of the authentication on the web is based. Asymmetric encryption algorithms such as RSA or DSA are used in Public Key Infrastructure [1] to generate key pairs. The key pair is composed of the so called public key and private key. As explained in the previous section, the mathematical relation between the two keys allows operations making infeasible, given one key, to find the correspondent pair. With this key pair two kinds of operations can be performed, encryption and signing. The first is usually done during a handshake of a security protocol (such as TLS or DTLS) to encrypt the symmetric key used to protect the rest of the communication. This is due to the fact that asymmetric operations are extremely computationally intensive, hence it is preferred to use it only for what is strictly necessary. The other possible operation is the signing operation. Given an amount of data (usually small, such as the result of a hashing process)

and a private key, it is possible to sign the data with the private key, so that when the data is received, who possesses the public key can verify that the data come from the peer that used the matching private key. If then, the corresponding public key is located in a certificate, and the certificate is trusted by a higher authority (that we in turn trust), it is possible to validate the authenticity of the sender.

In this document the emphasis is on an asymmetric algorithm using elliptic curves, called ECDSA.

ECDSA is a variant of the Digital Signature Algorithm (DSA) operating on elliptic curves. For sending a signed message from A to B, both have to agree on Elliptic Curve domain parameters. Sender A have a key pair consisting of a private key d_A (a randomly selected integer less than n, where n is the order of the curve, an elliptic curve domain parameter) and a public key $Q_A = d_A *$ G (G is the generator point, another elliptic curve domain parameter) [3]. The signed message is used for two main purposes in the TLS and DTLS handshake.

The first purpose of the signed message is to provide authentication: both peers (in case of mutual authentication) sign data with the private key. Then when the recipient receives the data, it can authenticate the sender with the public key present in the previously sent certificate. The second purpose is the key exchange, explained in the next section.

2.8 Key Exchange

During the handshake is exchanged a shared secret from which will be derived all the keys used to secure the communication between the peers. The shared secret is called pre-master-secret and it is usually exchanged with one of the key exchange algorithm. The pre-master-secret, combined with other parameters such as the two random numbers that client and server generated, is given to a pseudo random function to generate a master secret. From the master secret, depending on the cipher suite selected by the server, are derived the session keys used to secure the communication. This is done because the purpose is to initiate a secure communication on an insecure channel. Since the asymmetric cryptography is too computationally expensive to be used to encrypt and decrypt all the data, it is instead used to exchange in a reliable manner a symmetric key that will protect the data between the peers. This key can be already present on both peers (called in this case pre shared key), or can be exchanged using one of the key exchange algorithm, for example Diffie-Hellman to avoid key reuse. Diffie-Hellman uses the discrete logarithm problem to exchange the same secret between two peers. This is performed exchanging some parameters on which the peers agreed before the key exchange, in addition to other random values generated during the key exchange . Variant of this algorithm, such as the Elliptic Curves Diffie-Hellman (ECDH), perform the same operation using elliptic curves which makes the key exchange more efficient and robust exchanging instead of simple parameters, public keys.

The focus in this document is yet on another variant of this anonymous key exchange algorithm, called Ephemeral Elliptic Curve Diffie-Hellman (ECDHE). The main reason of the introduction of this variant, and its use, is that the peers have to previously agree upon certain parameters, that are always fixed (for instance in case of ECDH the public keys exchanged are the peer's public

keys used also for authentication). This goes against a property called Forward secrecy, that states that if a key is compromised, this will allow only access to data protected with that key, and it is not used to derive any other key [7]. In the ECDHE in fact, the server defines a point in the elliptic curve previously negotiated with the client, and uses it to generate temporary keys used only for the key exchange, granting forward secrecy and impersonation resilience. If in fact, during the key exchange, one of the key is compromised, it does not affect the authentication process since they are not used for other purposes. This is though more computationally expensive key exchange and requires more time to be performed, compared to the elliptic curve Diffie-Hellman. The obtained secret derived from this key exchange, it is indirectly used (since from the shared secret session keys are derived and used only within the session) in the symmetric cryptography.

2.9 Symmetric encryption and decryption

The key exchange and the authentication (thanks to the asymmetric cryptography), from the very last handshake message of each peer onwards, provide all the requirements to use the symmetric cryptography to secure the messages. Symmetric cryptography uses, as the name suggests, a symmetric key algorithm in order to perform encryption and decryption, in opposition to asymmetric cryptography.

Symmetric encryption is divided in block ciphers (that operate on a fixed amount of bits) such as AES, DES and Blowfish and stream ciphers such as RC4. Those ciphers can be used in several modes of operations; for example block ciphers can be used in Electronic Code Book (ECB) mode, Cipher Block Chaining (CBC) or Counter mode (CTR)[8].

The focus in this document is on the block cipher Advanced Encryption Standard (AES), used in CCM mode of operation, where CCM stands for Counter with CBC-MAC. Proceeding with order, the AES cipher allows, given a key of 128, 192 or 256 bits, to encrypt or decrypt blocks of 128 bits of data implying that if the data does not fit that precise amount of bytes, are padded (expanded with pseudo random data) before the encryption, and eliminated after the decryption so to not affect the plaintext but do not allow possibility to recognise pattern in the cipher text [8].

The CCM mode defined in the RFC 3610 [16] is the acronym for Counter with CBC-MAC and offers, differently for other modes, an authenticated encryption and authenticated decryption. The Authenticated Encryption with Associated Data (AEAD) is the interface in which ciphers such as CCM are defined [9]. The CCM mode allows in fact to use the same key for both encryption and authentication thanks to an initialisation vector. The encryption is performed thanks to the Counter mode [8] while the authentication is obtained with the CBC-MAC mode where CBC stands for Counter Block Chain and MAC stands for Message Authentication Code. The counter mode allows the transformation from a block cipher to a stream cipher (hence avoiding the use of padding), using an initial value (called nonce) and a deterministic counter function increased in every encryption block. The cipher text is then used as one of the parameter for the CBC-MAC together with a nonce, to generate the message authenticity code. The MAC is then postponed to the cypher text before sending the packet

to the other peer.

With this section it has been explain the last part of the main cipher suite used in this document, that corresponds to the following string:

TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8

where:

- TLS means that this is a TLS cipher (DTLS uses the same ciphers since the two protocols are very similar)
- ECDHE is Ephemeral Elliptic Curve Diffie Hellman
- ECDSA stands for Elliptic Curves Digital Signature Algorithm
- AES is the Advanced Encryption Standard
- CCM_8 stands for Counter mode with CBC-MAC with 8 bytes of associated data

Chapter 3

Secure CoAP for smartphones

In this section the current possibilities that can be used to achieve the objective of this thesis will be illustrated: how it is possible to secure the CoAP protocol. After this initial analysis the chosen option and its application in order to design and develop the solution will be explained.

3.1 Security solutions for the IoT

Before starting to design a possible solution, it is best practice to research possible ways to achieve the same options, since a shorter way to enable end to end secure communications in the IoT might exist. Currently, there are no APIs that allow the use of CoAP in a secure mode in Java language for smartphones. DTLS which is used to secure UDP communication is the transport protocol recommended by the IETF in the CoAP RFC, is available from APIs written in the programming language C, provided by OpenSSL [1] , CyaSSL [2] and GnuTLS [3].

One option is then the use of one of those libraries and connection of the CoAP implementation with one of these lower level libraries in the Android OS as security provider [4]. Alternatively it is possible to use one of the java security provider, without the need of connecting two different APIs that might introduce unnecessary complexity, or secure the communication in a lower level so to have "by default" a secure communication.

[1] OpenSSL is an open source toolkit implementing SSL and TLS protocols as well as a full-strength general purpose cryptography library. http://www.openssl.org

[2] yaSSL is an open source API that provides embedded security solutions with an emphasis on speed, size, portability, features, and standards compliance. http://www.yassl.com

[3] GnuTLS is a secure communications library implementing the SSL, TLS and DTLS protocols and technologies around them. It is aimed to be portable and efficient with focus on security and interoperability. http://www.gnu.org/software/gnutls/

[4] A security provider is as the name suggests a "provider" for security functionalities, such as algorithms, key generator, and so on. In the Android OS there are several security providers buit-in with the operating system which provide the most common security functionalities. In a Java virtual machine it is possible to find, among other providers, the Java Cryptography Extension (JCE).

It is necessary to make a small annotation before proceeding. According to the CoAP RFC [14] there are three ways to obtain CoAPS: with pre-shared key (in which case the symmetric encryption and decryption key is previously loaded in the peers), using raw public key (where the peers have a public key without certificate) or through certificates (X.509 certificates [4]). This is in addition to the no-security options, which is the plain CoAP protocol. The purpose of this document is not just securing CoAP, but allow a CoAPS enabled device to communicate not only with other smartphones, but also with constrained devices, such as sensors in the Wireless Sensor Network (WSN). Hence, the choice in the protection of the CoAP messages is more limited. In particular, there is the need to use a cipher suite that will not be too computationally expensive since it has to be executed in constrained devices. The recommended DTLS cipher suites in the CoAP RFC [14] such as AES-CCM are amenable to compact implementations, hence suitable for constrained environment [10]. The choice has hence to take in consideration this restriction.

There are several ways to proceed and in the following subsections are explained pros and cons of each of them, starting with the CoAP implementation.

3.1.1 Constrained Application Protocol

The objective here is to analyse if there are current CoAP implementation in the Java language, and that can be used in a smartphone. There are currently two java implementations of the CoAP protocol, and they do not take the security aspect in consideration. Those are jCoap [5] and Californium [6]. They both provide similar functionalities and can be used in the Android OS since they do not require additional libraries or particular setup, so both are possible candidates for the purpose of securing the CoAP protocol.

3.1.2 DTLS for CoAP

One possible alternative is to allow the CoAP implementation to send messages to the DTLS protocol instead of sending directly UDP datagrams. DTLS will then properly process the messages, securing them before the transmission. For the DTLS protocol there are two different possible ways to proceed: it is possible to use one of the C libraries, connecting it to the CoAP implementation, or use a Java DTLS implementation, allowing an easier connection with the CoAP protocol and the possibility to build a single application containing both CoAP and CoAPS capabilities.

Use of C libraries

A way to provide the CoAPS protocol is with the aid of the existing libraries, available in the C language provided by OpenSSL, CyaSSL and GnuTLS. Since Java is the primary and recommended language to develop Android applications, there is the need to address C code in Java using Java Native Interface (JNI). This increases the complexity of the application and reduces the portability.

[5] jCoAP is a Java Library implementing the Constrained Application Protocol (CoAP). http://code.google.com/p/jcoap/

[6] Californium (Cf) is a very modular CoAP framework written in Java using a flexible, layered architecture. It allows easy creation of cloud-based client and server applications as well as CoAP proxies. http://people.inf.ethz.ch/mkovatsc/californium.php

Moreover it is specified in the Android NDK documentation that using native code does not result in an automatic performance overhead, but increases the application's complexity. For this reason it is appropriate to use the native libraries only whenever necessary.

Even if Android is provided with the built-in OpenSSL libraries, the link agains system libraries is discouraged because there are different existing versions and depending on the platform and version the compilation (and linking) may vary resulting in errors. It is recommended to include the ARM-compiled version of the C libraries together with the application that uses them to avoid such problems. This will increase the complexity of the application, since it is needed to develop two applications: one consisting of the C libraries, and another one in Java that includes the main application and the call to the libraries before specified using JNI.

Java implementation

The alternative is a new implementation of the DTLS protocol resulting in a whole program written in the Java language. This grant an easier development and debug, and increases the compatibility and portability, since the Java byte code is executed by a virtual machine that works in heterogeneous environments. It will furthermore provide an alternative to the use of C libraries, currently missing.
This can be obtained in two ways:

- Implementing from scratch the whole DTLS library in Java language
- Modifying the current TLS implementation [7] introducing datagram communication with other functionalities needed for the application

With a new implementation the functioning will be straightforward to achieve the requirements, it will be lightweight since it will be implemented only what is needed. This however might introduce new bugs in comparison with an already tested API. Also given the security obtained with the algorithms used, it is important to use them in the correct way in order to not nullify their strength.

Since the DTLS protocol illustrated in section 2.3 page 9 is very similar to the TLS protocol (illustrated in section 2.2 page 5) the implementation of DTLS from the existing TLS will give an easy and portable solution entirely written in Java, without the need to include any external library (below explained in details) and use of JNI.

3.1.3 IPSec for CoAP

One of the possible alternatives to the implementation of the DTLS protocol is the use of IPSec that operates at the network layer and the CoAP protocol, that works at the application level, so to secure all the traffic by a lower level in the OSI model. IPSec however is an end-to-end protocol and it not optimised to address applications independently. In section 2.1 of [12] is shown that also the key management and the security policies are not automatic but must be created by administrators or hardcoded in host-based implementations. It might not be possible to access a deployed sensor, or it might be

[7] Provided by one of the Android security provider

needed to change applications or security permissions dynamically, hence the use of IPSec is discouraged. A system that has an automatic key management such as PKI is preferred. Moreover IPSec is typically implemented in the kernel hence it might not be supported by many embedded IP stack; firewall and NATs may furthermore limit the use of IPSec [14]. Regarding the compression, since the communication will be mostly between a smartphone and a sensor in a 6LoWPAN environment, using a lower protocol for the encryption will decrease the efficiency of the compression since it is not possible to compress encrypted data. Using higher level protocols in which only the payload is encrypted, the compression will be more efficient.

3.1.4 TCP encapsulation

Another possible way to reach the goal of securing the CoAP protocol is to encapsulate CoAP in the TCP protocol. Even though CoAP is designed for a non reliable protocol such as UDP, it is possible to encapsulate it in a reliable protocol. The TCP encapsulation will allow the possibility to use the widespread TLS protocol, a standardised solution available on a large number of operating systems and devices. Since TLS is an application level protocol it is easy to deploy and manage, it is hardware independent and it doesn't require kernel functionalities in opposition to IPSec. TLS also integrate a mechanism of key management (as explained in section 2.2 page 5) making it suitable for inaccessible nodes. In this scenario CoAP will need to send only (*non confirmable*) (section 2.1 page 4)) messages obtaining reliability automatically, since TCP will automatically handle retransmissions, packet duplication and reordering. However, if there is the need of real time communication (i.e. retrieve data from a sensor) in a constrained node, the overhead caused by the TCP protocol might be too high to handle the traffic, generating excessive delay, since it is not designed for constrained environments. A more lightweight protocol such as UDP that allows fast communication is necessary, at the expense of taking care of some network aspects at application level instead of automating them in the network layer.

3.2 CoAPS for Smartphones

Starting with the CoAP protocol, the choice is between jCoAP and Californium. Since both implementations are similar, they are both written in java and portable to the Android OS, both can be used. Californium though is more up to date with the continuous changes in the CoAP RFC, and has a built-in mechanism of CoAP resources, missing in jCoap. Moreover the underlying protocols are handled in a slightly better way than jCoap, facilitating the connection with the DTLS implementation, thus making it the preferred choice.

Regarding the selection of the protocol to secure CoAP messages, as above shown, there are several ways and all of them are feasible, but it is important to find a solution that suits most the objective of this document.

The TCP encapsulation has been excluded since the TCP protocol will introduce too much overhead. This, in addition to the TLS overhead, making it not suitable for constrained networks and nodes. It is moreover not always

convenient to communicate with the TCP protocol on lossy networks and have fast delivery.

The use of the IPSec protocol has also been discarded; since one of the end point might be a sensor, and it might contain more than one application with different access rights, using IPSec will prevent addressing the single process. It is possible to avoid this behaviour with policy files with IPSec, but the device needs to be accessed in order to change these policies. In some cases it is not ideal to require access to sensors, since they might be located in not easily accessible environments, and it is not ideal to require continuous maintenance. It is preferred instead a method with an automatic access and key management.

The remaining choice is either to use already existing libraries or implement the DTLS protocol from scratch. It is always good to avoid to reinvent the wheel, so first it has been analysed the possibility to use existing libraries. As explained above, the current DTLS implementations are written in C, there is at the moment no Java library that offers DTLS functionalities. The Android operating system has built-in some basic cryptographic functionalities, provided by its security provider, such as OpenSSL. These providers offer to the java environment the possibility to use several security protocols, but in this case there is the restriction related to the cipher suite illustrated in section 3.1, so they cannot be used. It is hence possible to implement required cipher suites in OpenSSL and embed this API with the CoAP application, inserting in Java the newly modified security provider, and bundle it together in the Android application. The downside here is that C implementations are hardware specific, and even without the use of particular libraries, the code needs to be recompiled in every hardware different from the building environment. This creates compatibility problems, but this option is valid unless it is found a better solution.

Considering the fact that DTLS and TLS are very similar, and the non existence of DTLS java implementation, it is possible to "extend" the TLS protocol with the required functions so as to reduce time and the amount of new code to be written. In the java environment the TLS protocol is implemented in the Java Cryptography Extension (JCE) or by the Bouncy Castle security provider. The JCE offers at the moment almost all the algorithms and cipher suites needed, but they are not available in Android since the latter possesses a subset of the Java functionalities. Bouncy Castle on the other hand is an API appositely designed to provide cryptographic capabilities to the Java environment, hence seems to be a valid candidate. It does not has all the prerequisites needed in order to provide the required cipher suite, but the effort needed to add these functionalities is minimal compared to the one needed to implement the whole DTLS protocol and cipher suite needed. Even though it is not possible to use Bouncy Castle as it is on the Android environment, there is a repackaging called Spongy Castle, that is designed to work on Android. Since it is needed to modify the security provider, it is not possible to use directly the library as jar file (as it is commonly done). After the proper modifications to the sources, it is possible though, to release a patch for the Bouncy Castle, then repackage it with Spongy Castle and then use the jar file to provide the desired DTLS protocol with the proper cipher suite needed for this scenario to protect CoAP messages. The downside here is that Bouncy Castle (and hence Spongy Castle since for this purpose it is possible to consider them synonymous) have only a TLS client, and no server, so there is much work to be done to have the DTLS protocol with the required cipher suite, but still it is possible to use several cryp-

tographic algorithms avoiding the implementation of the whole protocol from scratch.

The last one seems the most reasonable and fast solution to reach the goal. Using the C libraries will require less code to be written (since they already implement the DTLS protocol and partially the required cipher suite), but it might cause compatibility problems and it requires anyways to embed an external library. Developing the whole application in java, will result in a more compatible environment (if for instance sensor nodes can run java, the CoAPS algorithm is already ready to deploy) and it will require in the same way to embed the cryptographic library. For these reasons the selected way to proceed is to implement the DTLS protocol and the required cipher suite in the Bouncy Castle security provider implementing the missing functionalities. In the remaining of this section it will be explained the method used to reach the goal, now that is clear the way to proceed.

3.3 Phases to enable CoAPS for smartphones

The work is divided in three different phases, and each phase requires the completion of the previous one. It is hence possible to position milestones at the end of each phase in order to monitor the progress of the work.

3.3.1 First Phase

The first phase is information gathering, studying of the related protocols and building a complete theoretical schema of the messages and functioning of the application in object. This will be useful for two reasons: first it will give an exact and visual understanding of all the network messages needed to make available the API for the CoAPS protocol, facilitating the development. Moreover, having a precise schema of all the interactions, allows to check step by step the reliability of the implemented solution, in order to detect in advance and reduce errors. The Android software development kit is then analysed to allow the development of the application in order to design the API, and the availability of CoAP libraries to allow the CoAPS protocol. The completion of the first phase gives precise information on the use of cipher suites in Java and in Android and a detailed and complete schema of the messages and actors needed to fulfil the objective.

3.3.2 Second Phase

With an understanding of the protocols, development environment and tools needed, it is possible to proceed to the second phase. The next step consists of selecting the most suitable way to reach the goal, and implement the chosen solution, discussed in this chapter. The implementation is carried on with an iterative and incremental development, without fixed iterations but more oriented to a SCRUM process.

The second phase is divided in several sub-steps. Once decided the way to follow, the problem is subsequently decomposed in smaller and easier problems. Unfortunately most of the functionalities require a working protocol, even with basic functions, which is at the moment not available. It is best practice to start

from the most critical part, the core of the application, in order to face potential problems and receive a feedback beforehand, which in this case is the DTLS handshake protocol. This step is also motivated by the limited documentation regarding both DTLS and CoAPS. At the moment of the writing the DTLS protocol is at the version 1.2 released at the beginning of the thesis, and it is not yet implemented by any security provider or library. The CoAP protocol is instead an Internet Engineering Task Force (IETF) draft and is subject to continuous modification.

Initially a client server application will be implemented using the existing libraries on a desktop or laptop, using libraries that can be used on the Android OS. From this point on it will be straightforward to proceed with the next step that is adding more functionalities to finally reach the required cipher suite and the secure transfer of CoAP messages. Successively the DTLS protocol is connected to CoAP.

3.3.3 Third Phase

The third phase is the evaluation of the implementation which will give a practical understanding of how feasible is to use the CoAPS protocol. This is obtained thanks to the comparison of CoAPS with the insecure protocol (CoAP). A slowdown in the communication is expected but it is unsure if this will heavily affect the performances, prohibiting its use in real applications.

The variables and parameters important for the system evaluation will be first analysed, then a reliable way to perform the measurements, and finally the results will be presented in tables and graphs to have a visual and immediate understanding of the comparison.

Those three phases can be schematised as following:

Phase 1: Research on existing documentation

- Review TLS, CoAP, 6LoWPAN, DTLS and related protocols
- Design a theoretical model for the DTLS communications (protocol and fields)
- Analyze current DTLS APIs available in C only
- Analyze current CoAP APIs
- Research Android application development

Phase 2: Build the solution

- Develop a client server application of a basic DTLS protocol
- Enhance the previous implementation iteratively to fulfil all the requirements
- Execute the application in the previous step in Android OS

- Modify a CoAP API to allow the use of the application from the previous step in the *coaps://* protocol

Phase 3: Evaluation

- Determine key parameters to evaluate
- Efficiency evaluation: memory, resources, response time and overhead

Chapter 4

Design and Implementation

This section illustrates the design and developing process of the Android application. Initially the development setup used to begin and carry on the implementation will be explained. Implementation choices will be then shown and motivated, followed by the design and implementation of the DTLS protocol starting with the handshake, defining a skeleton and adding iteratively more functionalities until there are all the required functions. The last subsection is the connection between the DTLS protocol and CoAP

4.1 Development Setup

First of all the development environment that will be used to build the application needs to be decided. This is quite straightforward. Since the final product is an Android application, the Android Software Development Kit (SDK) provides API for development, testing and debugging and it is very well integrated with the Eclipse IDE. There is a plugin called Android Developer Tools (ADT) for Eclipse that provides even graphical interface assistance for the design of the user interface.

In order to run and debug the application there is the need to install it either in an emulator or in a mobile phone. The Android emulators are quite slow, especially during the startup, and they do not provide a stable environment to run applications. Moreover the implementation will require to be run in separate computers for reasons later explained. The ADT provides a way to interface an Android phone directly with the computer, allowing to display debug and log messages directly in eclipse. ADT offers also the possibility to debug the application in eclipse following the program step by step, in the meantime it is run on the mobile phone, connected via a USB cable. This is the desired developing setup; every time it is needed to run or debug the application, Eclipse will automatically compile it and send it to the android phone, installing the application that the ADT has already built, and then finally run the application itself.

4.2 Implementation Choices

The first objective is to allow a CoAP-enabled smartphone to communicate in a secure way; the most appropriate way is through the use of the DTLS protocol [14]. In order to provide this possibility in a broad manner, the main purpose of the implementation is to provide an API so any CoAP implementation will be able to transmit and receive secure messages. Moreover, since the environment is in Java, future sensors or other devices able to have a similar java virtual machine as the Android phones will have the same possibility; and this can be used not only for CoAPS but for more general and broad purposes.

During the programming phase there are decisions to be made in order to organise the code in a structured and legible way. The very first is relative to the way in which data are handled. Java is an object oriented programming, hence data must be divided in objects, and each object contains all the data related to itself. Now, there are two different ways of storing data; in this scenario data are transferred in bits from one peer to the other, since they need to be transferred through a channel, and they also need to be serialised. This data can be stored in two possible ways: either storing in the objects the bits containing the data, or converting the data in primitive java types and store them in the objects. In order to perform operations though, it is needed to convert this data from bytes to primitive types; this process is done in an early stage and only once if the data is parsed during the object creation, or they can be parsed just before that exact data it is needed (with the problem that if the data is needed more than once, it will be necessary to parse again the data or store it).

The implementation choice made here is that each data is parsed immediately: as soon as a (DTLS) message arrives, it is parsed and create the relative objects depending on the message, containing primitive data. Before sending the next message, there is the inverse parsing, that transforms the data contained in the objects in bytes, serialised and ready to be sent to the other peer. This will give also the possibility to understand what is happening in the protocol, since all the data is readable since it is primitive types. This is true for almost all data; in some cases, for instance during the prime number generation, or hashes, it is more convenient to store directly the byte value, in order to avoid parsing when it is not needed.

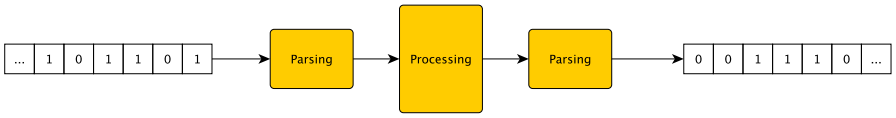

Figure 4.1: Data Parsing and processing

4.3 Creation of the development environment

Before starting the implementation it is necessary to set up the development environment. As above explained the Eclipse IDE is required, with the Android plugin. Also, after the project creation, it is needed to download the source code of the Spongy Castle security provider, since the implementation will reside inside this security provider. Following the same naming methodology, the

root package name will be *org.spongycastle.crypto.dtls*. This package will contain all the sub packages with the code necessary for the DTLS protocol, such as constants, core, ciphers, key exchange, context, and so on. Having the sources available, and the TLS client, it is easy to reuse the existing code and conventions, to avoid repetitions and unify the implementation with the existing TLS handshake whether possible.

4.4 Iteration 1: DTLS Handshake

The first iteration aims to implement the DTLS handshake with the null cipher suite DTLS_NULL_WITH_NULL_NULL. This is in fact the cipher suite used in the first phase of the full DTLS handshake [1], when there is still no encryptions and the parameters are exchanged in plain text. This phase is the prerequisite for all the other cipher suites, hence this is the implementation step.

At the end of the iteration 1 the result is a client and server, capable to perform a DTLS handshake with the *null* cipher suite. This is mainly useful for two reasons. First this cipher suite does not provide encryption, but provides integrity. Both peers keep a hash of all the messages [2] that must be confirmed from both sides with the *Finished* message. If those two hashes are not the same, the handshake will fail; this means that a data alteration between the two peers has occurred. And for the properties of the hash if there is even one bit different, the hash will be different.

Here not all the handshake messages are implemented, since not all of them are needed for this negotiation. In particular, all the optional messages have been discarded, such as the *Certificate*, the *ServerKeyExchange* and the *CertificateRequest* for the server in the flight 4, and *Certificate* and *CertificateVerify* for the client. Moreover additional extensions (as explained in next iterations) for *ClientHello* and *ServerHello* messages are ignored since not required.

Other simplifications regard the creation and handling of the cookie for the "hello" messages (that has the purpose of preventing DoS attacks), session resumption (the server maintains an internal state in order to abbreviate successive renegotiations with the client) and in the first phase the final check of the messages' hashes in order to control the integrity of the handshake's messages. Also the handshake messages can be fragmented, in case they exceed the maximum length of one datagram, and they need to be fragmented. This feature it has been ignored as well in this iteration. It is possible to send each DTLS record in a single datagram, or multiple records in the same datagram; in this iteration only the second case will be considered, in which when more records compose a single flight (flights 4, 5 and 6) will be sent in the same UDP packet.

The first operation to perform, according to the implementation choices, is to implement the parser. This object will be in charge of serialising and deserializing every field in the messages in different protocols, in order to be transmitted and received. The communication is otherwise impossible. It is worth reminding that the DTLS protocol, as TLS is composed of a "carrier" protocol, called

[1] The *null* cipher suite it is used from the very first message, till to and included the ChangeCipherspec message. That message informs in fact the other peer that from the very next message it will be used the cipher suite just negotiated.

[2] The First ClientHello sent from the client, and the HelloVerifyRequest are not considered in this calculations, since they are part of a mechanism that prevents DDoS attacks, because of the susceptibility of the UDP protocol to this kind of attacks.

Record Layer, a protocol that handles fragmentation, called *Fragment*, *Handshake* containing the handshake messages, and finally the *ChangeCipherSpec* to signal that the peer is changing cipher suite in the next message, and the *Application* to transport application data. Gradually, after the implementation of the parser for the messages, also the several classes will be created, such as a class representing the Record Layer, that will contain all the other protocols. Each class contains all the required fields, whenever possible using primitive types, and getters and setters. It will also contain a static method for the creation of the object, completed when the object is created for the first time. When the object must be deserialized, after proper parsing and transformation into primitive types, the fields are populated thanks to getters and setters.

The macro object of the DTLS handshake is a class called *DTLSProtocolHandler*, that gives the possibility to a client and server to use the protocol, and houses the basic logic. The state machine of the DTLS handshake is described in the RFC 6347 section 4.2.4 which defines four states:

- Preparing: the implementation performs whatever computation necessary to prepare the next flight of the messages and put the messages in the buffer ready to be sent
- Sending: the implementation transmits the buffered messages; once the messages have been sent, the implementation enters in the Finished state (if the last message has been sent), or in the Waiting state
- Waiting: in this state the implementation waits for the next flight to be received within a certain amount of time. If nothing is received within a certain time, it resends the last buffered messages; the same thing happen when a retransmission is received. If instead a regular flight is received, the next state is the preparing state, in which the next flight is prepared, otherwise if the last message has been received, the implementation transits to the Finished state.
- Finished: the implementation receives the next flight; if it is the last flight, than the handshake is completed, otherwise it returns to the preparing state (for instance in case of the server in the handshake, it receives the client Finished message, but the server still need to send the ChangeCipherspec and Finished).

This is a sequential process, that is performed, in this first iteration, in a single thread, using blocking calls (since the next message cannot be sent until all the required messages are received).

In this process it is easy to understand that there are several queues, in order to buffer messages, keep partial hash of the messages, and to store application data. If, in fact, the application wants to send data but the handshake has not yet been performed, the data is stored in a buffer, and send as soon as the handshake has been completed.

Constants such as the retransmission timer, reside in an apposite package, together with other constants such as the protocol version, and implementation constants defining cipher suites, UDP port, and so on.

Once the machine state and the parsing are ready, empty Records are transmitted and received so to validate the implementation and reach the finished

state, that determines the end of the handshake. Since the finished state is not a proper end of the handshake (since the server needs to transmit its last flight), two additional states have been introduced, the *Handhsake Not Started* to determine when to buffer data and start the handshake, and the *Handshake Finished*, meaning that from this moment, the protocol can encrypt and send application data, and the handshake protocol is no more involved. With this additional states, it is possible to determine, when a data must be sent, if the handshake must be performed, or if it is already completed.

The next step is to populate all the previous messages with real data, such as random number representing server and client random, and create a *context* in which the peers store data received, partial hashes and so forth. The finished messages, in fact, requires a hash validation. This is obtained performing the hash of the Handshake protocol, as it is not fragmented. Before obtaining the hash it is needed to reassemble the handshake message, which is not taken in consideration in this iteration, and update the hash value every time a message is sent or received. The first *ClientHello* and the *ClientHelloVerify* are not taken in consideration for the hash computation. The hash algorithm used for this cipher suite is the SHA 256, in preparation for the use in the final cipher suite and a standard for the DTLS protocol version 1.2.

This implementation aims, in fact, to have a functional DTLS version 1.2, that uses instead of the MD5 SHA-1 hash combination, the SHA 256 algorithm. In particular, in the finished message the simple hash of the messages is not sent. This result is one of the input for a function called Pseudo Random Function (PRF). This function was not yet implemented in the Bouncy Castle provider, hence it has been implemented afresh. This function takes, in addition to the final hash of the messages, the master secret and a label (that in this case is the conversion in byte array of the string "client finished" in case of the client, and "server finished" in case of the server)[3].

The master secret is generated combining the pre master secret, the client random, the server random and a label (that is again the conversion in byte array of the string "master secret"), as input to the PRF function. The pre master secret is in this instance null (since there is a null key exchange), but in the next iterations will be the shared secret generated with the key exchange. The result is called *Verify Data* and this is the value sent in the **Finished!** (**Finished!**) message, so not only validates all the messages, but also that the key exchange happened in the correct way, and if even a bit in this process does not correspond to both sides, the handshake will not complete.

Once this is functional, it is possible to send and receive plain text application data, after the DTLS handshake that uses the cipher suite:
TLS_NULL_WITH_NULL_NULL [4].

[3]So in the finished message, both client and server have to use the final hash twice; once to compute its own hash, and to verify the received hash.

[4]This cipher suite must be disabled in TLS and DTLS since it does not provide any privacy. A client might force the server to handshake with a low security cipher suite, making possible to eavesdrop the application data. It is though useful for debugging and testing purposes.

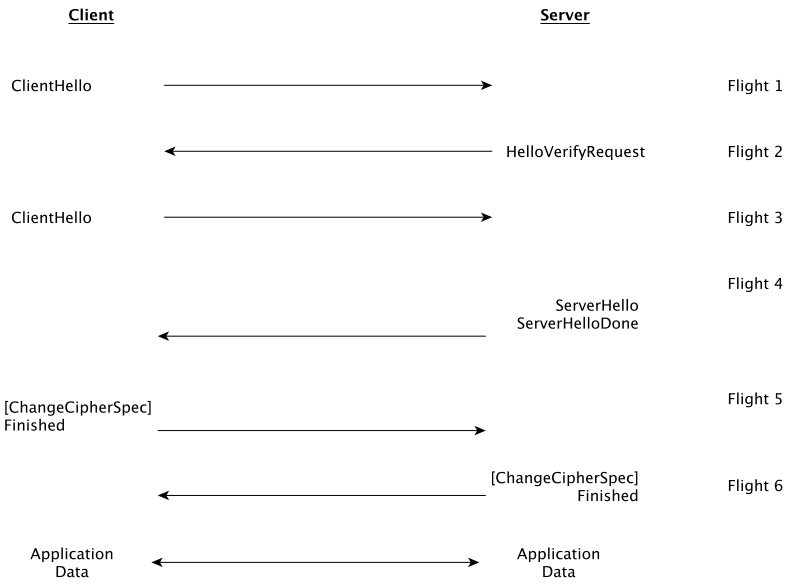

Figure 4.2: Handshake messages for iteration 1

Figure 4.2 shows the handshake messages available thanks to this iteration. The cipher suite just implemented must be used only for development or debug purposes, and cannot be use in production environment. This handshake is though useful in this scenario since, following the iterative programming paradigm, allows the creation of an initial prototype with basic functionalities, that can be enhanced in future iterations. It is much easier to add functionalities to an existing system, making it more and more complex, rather than start with all the functionalities from the first messages. This is the main advantage of iterative and incremental programming.

4.5 Iteration 2: DTLS with pre shared key and symmetric encryption

With the second iteration the possibility to have a pre shared secret between client and server and send and receive encrypted messages is introduced, thanks to the cipher suite AES_128_CCM_8. This requires a different initial setup, since the handshake has now the prerequisite to have a shared secret, previously exchanged between client and server [5]. After refactoring the client and the server, as well as the main class containing the logic, it is possible to implement the key exchange and expose the new cipher suite in the client and in the server. Since now there are more than one cipher, there is the need to implement a mechanism of cipher suite selection during the handshake. The client offers,

[5]This is usually done using a different channel, for instance it is set up during the installation, or manually, in order to prevent eavesdropping since there is no secure communication before the fist handshake is completed.

in the ClientHello, the available cipher suites. The server selects the strongest possible supported by the client, transmitting it in the ServerHello message.

In this case the pre master secret is not null anymore, but it contains the shared secret. This secret is though never sent through the channel, but the client and the server have an ID, that is the same for both peers, representing the identifier of the shared secret that it is going to be used in the handshake.

The handshake messages are substantially the same as the previous iterations, excluding few extra fields and the last messages of the last two flights. The logic is slightly different though, since there is the need to build the session keys, derived from the master secret, used for the encryption and authentication of application data.

After the *ChangeCipherSpec* messages, in fact the peers must compute the session keys, that will be used from the Finished messages on. In this scenario, there is a small detail to take into consideration: it is not possible anymore to parse all the messages at the same time. This is true for the flight 5 and 6. The reason is that there is a mechanism to differentiate messages before and after the *ChangecipherSpec*, called *Epoch*. The epoch in fact determines which state must be used in order to parse and decrypt the messages, since, in case of the first handshake, the following messages are encrypted, differently from the one before the *ChangeCipherSpec*. During the flights 5 and 6 the DTLS must queue messages with a greater epoch than the current, and once it is possible to switch the state, process the pending messages. In this phase DTLS has in fact a reading state and a writing state in order to properly handle this particular case.

The session keys are generated in the following way: the pseudo random function it is used with inputs the master secret (obtained as in the previous iteration, but with a not empty pre master secret), the concatenation of the two random number generated by client and server, and a label (that is the conversion in byte array of the string "key expansion"). The length of the result depends on the selected cipher suite, hence must be calculate dynamically. The size is varying according to the length of the encryption key, that in this case is 128 bits since the encryption algorithm used is AES 128. The client and server initialisation vector (used in this cipher suite) and MAC of client and server (that are not needed for the CCM mode hence not taken in consideration). The long result of the PRF is then divided in smaller byte arrays, resulting in: *client_write_key*, *server_write_key*, *client_write_IV* and *server_write_IV*.

All the required inputs to perform encryption are now available. The AES_128_CCM_8 is though not implemented anywhere in the Java environment, so it needs to be implemented. Bouncy Castle has both the AES algorithm and the CCM mode, so it is needed to use them together with associated data, nonce and initialisation vector, in order to perform the authenticated encryption and decryption.

Associated data are based on DTLS record's parameters such as epoch, sequence number and DTLS version; the nonce is the concatenation of the epoch, the sequence number and the previously generated initialisation vector, while the explicit IV is the concatenation of epoch and sequence number.

The authenticated encryption is obtained given the encryption key (previously generated), the nonce, the associated data, and the plaintext. The result is a cipher text, with the same length of the plain text with additional 8 bytes (since this is CCM_8) composed by the length of the MAC and the explicit IV

(needed for the decryption). If the decryption does not have the exact same data, or there is any alteration during the transmission, the decryption will fail.

Sending and receiving application data, confirms the correctness of the handshake and the encryption, that has been tested with another existing C application called TinyDTLS [6] for compatibility, confirming the functioning of the cipher suite TLS_PSK_WITH_AES_128_CCM_8.

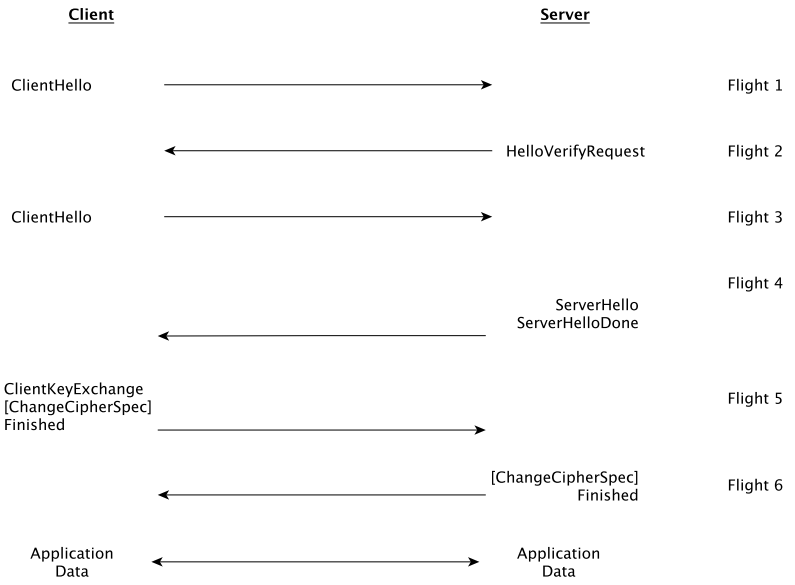

Figure 4.3: Handshake messages with pre-shared key

In the handshake messages of figure 4.3 has been introduced data encryption with a key that is set up on the peer before the handshake starts. Now application data exchanged after the handshake is encrypted using the AES algorithm with the key shared between the peers. This modification caused some changes to the previously implemented exchange algorithm in iteration 1, which have been refactored keeping the code as clean and easy as possible to easily add new functionalities.

4.6 Iteration 3: DTLS with ECDHE key exchange

The pre shared key might be inconvenient in some situations; first of all the possibility to reuse the same key should be avoided, because it might be compromised and might help the attacker. With this iteration it will be introduced a variant of the Diffie-Hellman key exchange, based on asymmetric cryptography and elliptic curves.

The Diffie-Hellman key exchange allows two parties to exchange a key based on a shared secret, that is in TLS and DTLS transmitted in previously hand-

[6] http://tinydtls.sourceforge.net

shake messages, and it is based on the discrete logarithm. A more efficient way is to perform the key exchange using elliptic curves, as explained in section 2.7 page 14. The Elliptic Curve Diffie-Hellman (ECDH) allows, in fact, to perform the key exchange with asymmetric cryptography generating a key pair based on a point (selected by the server) on a previously negotiated elliptic curve. This can be obtained if the peer has a certificate with an elliptic curve key used as shared secret, that will generate a pre master secret on which future keys are derived. This method does not offer though future secrecy, since the same shared secret is used in more than one handshake (even though the point on the elliptic curve selected by the server changes over time).

A stronger way to perform the ECDH is to generate on every key exchange a key pair (operation done by the server) and use that shared secret to perform the Diffie-Hellman. This is obviously more computationally expensive than using the certificate's key, but it prevents the reuse of the same key since it is generated every time. This is the key exchange that is needed for the cipher suite that is going to be completed in the next iteration, and the recommended way according to the DTLS RFC.

In order to perform this key exchange, it is required that the server has a certificate, since its private key will be used to sign the hash of the parameters transmitted in the *ServerKeyExcange*.

A certificate hierarchy is then build, with a self signed certificate as top Certification Authority (CA), and two end point certificates, signed with the top CA, that will be given to the server and to the client (the client has a certificate if it is needed mutual authentication with certificates; this is optional). This is a simplified schema that works as proof of concept; in real applications the organisation might pay the CA to issue certificates. For this purpose this was not needed since in order to verify a certificate, the "signing path" is checked to determine if it leads to one of the top CA's certificated locally stored, and if it is stored the self signed certificate, it is possible to validate signatures.

The generation of the certificate hierarchy must be performed before the handshake takes place. The certificates and the private keys are stored in a key store, in order to be read by the DTLS protocol. It is then needed a refactor of the DTLS server that now needs, in the initialisation phase, to retrieve and store the certificates in its context, to be used during the handshake.

Once the previous steps are implemented, it is possible to proceed with the *ServerKeyExchange* handshake message. The server must select a point in the elliptic curve (from which the key exchange public key is obtained and sent) and the parameters specifying to the client which type of elliptic curve has been used. The server will then sign the public key with its private key, so that after the client has received the server's *Certificate* is able to determine the integrity.

If the previous steps and the signature are validated, the client is able to generate a key pair and calculate from that the pre master secret. It will then send its public key so that the server will generate the same pre master secret, and proceed with the *ChangeCipherSpec* now that both peers are able to generate session keys based on the same pre master secret obtained with the key exchange.

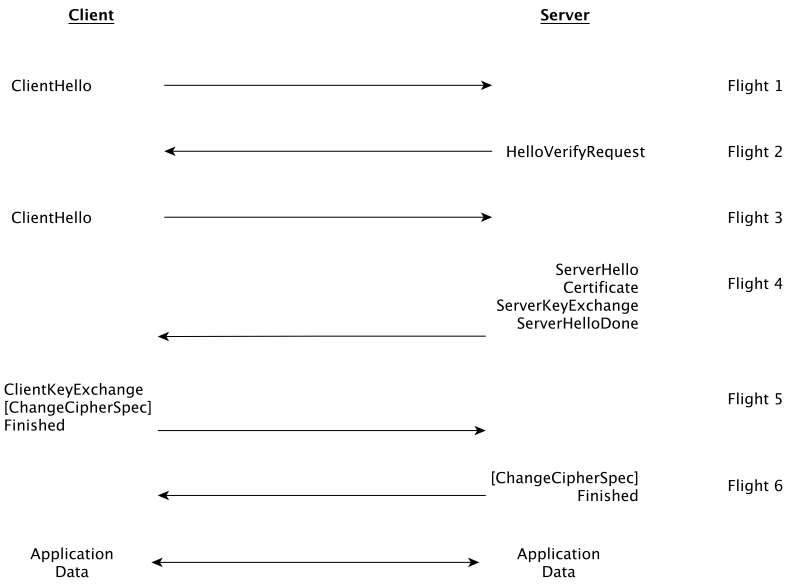

Figure 4.4: Handshake messages with ECDHE key exchange

Figure 4.4 shows the handshake messages for the third iteration. This iteration brought the implementation close to the final stage, since the ECDHE key exchange required the introduction of several new messages (among which the ServerKeyExchange), and also the improvement of existing ones (such as the introduction of extensions for ClientHello and ServerHello messages). These messages are used in the next and final iteration.

4.7 Iteration 4: DTLS with mutual authentication

In the previous iteration a certificate hierarchy has been generated. The certificates contain an Elliptic Curve Digital Signature Algorithm (ECDSA) public key that the server uses for the ECDHE_ECDSA key exchange. In order to perform mutual authentication, in which also the client has a certificate and a private key (so that not only the client can authenticate the server, but also the client is authenticated) needed to allow the DTLS client to access its own certificates and private key.

To have mutual authentication, the server will send a *CertificateRequest* message to the client, and the latter will reply with a *CertificateVerify*.

The *CertificateRequest* message, immediately sent after the *ServerKeyExchange* message, contains the supported hash algorithms and signature algorithms that the server is able to verify, and the list of Distinguished Name (DN) acceptable (or an empty list if all DN want to be considered valid). The client *Certificate* contains the end user certificate of the client, that the server must validate to authenticate the client.

After the client *certificate* message, the *CertificateVerify* is sent. The latter

sends a signed hash of all the previous messages (excluded the first ClientHello and the HelloVerify). The hash is computed with a different algorithm than the one used for the *Finished* message (verify_data) hence it is needed to refactor the hashing procedure to update both hash algorithms during the handshake. The result is then signed with the client's private key (needed for mutual authentication) and verified by the server. The latter does not only verify the signature using the client's public key obtained in the previous message (client Certificate), but it has its own hash that verify against the one provided in the CertificateVerify. In this way the client can be authenticated, proving that it is the owner of the certificate previously sent (since it is not possible to verify signatures with a public key, not signed with the corresponding private key).

This brings to the final cipher suite:
TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8

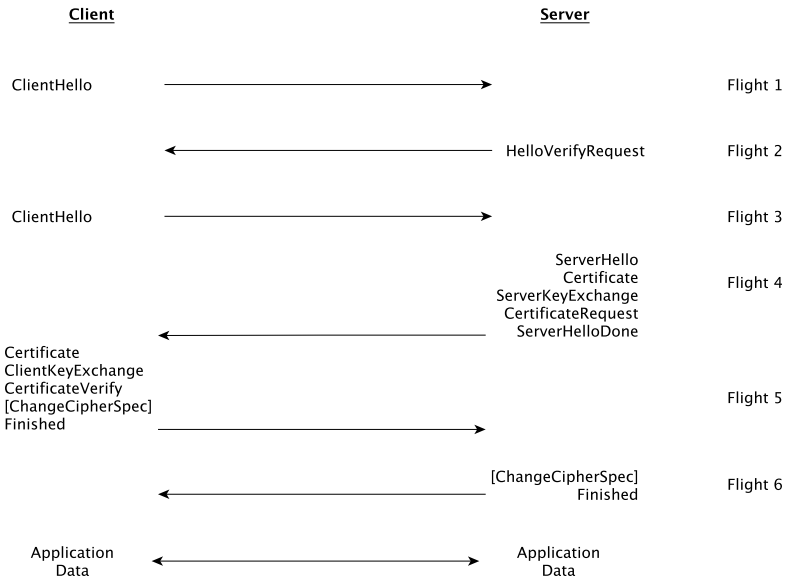

Figure 4.5: Handshake messages for the final cipher suite recommended by the IETF for the CoAPS protocol [14]

4.8 CoAP and DTLS integration

Once the DTLS handshake and the secure data transmission is possible, the next step is to allow the CoAP protocol to use it, when sending CoAPS messages. The CoAP protocol should be able though to send both secure and insecure messages. In the implementation this is translated in having two UDP ports listening, respectively on the port 5683 for CoAP and 4433 for CoAPS.

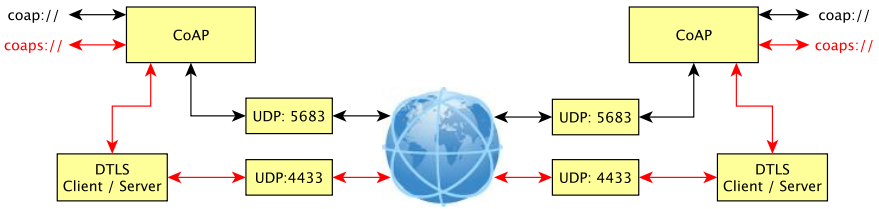

Figure 4.6: CoAPS for smartphones

Figure 4.6 shows the existing CoAP protocol, already implemented in Californium (with black lines), and the new CoAPS implementation using the DTLS protocol (in red). Californium uses a Publisher-Subscriber pattern to notify the reception of a new datagram to be processed. This is necessary because CoAP messages can be asynchronous, and multiple requests can be sent before receiving the response; moreover CoAP allows the server to respond to a confirmable message with an acknowledgment message before sending the actual response. The DTLS handshake, must instead work in a synchronous way, since it is not possible to send the next handshake message if the previous ones has not been received. This is not valid for application data though.

It is then needed a refactor of the datagram processing, in order to make DTLS work in a synchronous way during the handshake, and process asynchronously application data. To obtain this behaviour the whole reception part of DTLS has been refactored. A receiving thread is continuously listening for incoming datagrams. As soon as a new datagram is received an event is published: the *DTLSProtocolHandler* (the DTLS macro object) catches it and processes. The processing can happen in two separate ways: during the handshake datagrams are stored in queues and processed sequentially thanks to the sequence_number and message_sequence fields present in the record layer and in the fragment; as soon as the handshake is completed, datagrams are not queued anymore, but processed directly. Once the message has been decrypted and decompressed by DTLS, another event is published, and the subscriber is this time the class that connects DTLS and CoAP.

The CoAP protocol, has in fact, now to be initialised with new properties, in order to enable CoAPS. The CoAPS client and server must be initialised with properties and files needed for DTLS such as key stores containing certificates and keys. The first time that the application wants to send a CoAPS message, Californium will, after processing the message, deliver it to DTLS that will perform the handshake and then send the CoAPS message. The recipient will then receive it on the CoAPS port (different from CoAP to distinguish the two protocols) and successively decrypted and processed.

Chapter 5

Evaluation

This section illustrates how the evaluation of the DTLS protocol has been executed. The environment in which the tests are done will be first described, then which and why those tests were chosen and the actual result of the tests, partly shown in graphs and part in tables.

5.1 Testing environment

As previously described, one the final objectives of this work is to enable the secure CoAP communication from a smartphone to a sensor node in a wireless sensor network or other devices. There are currently no available CoAPS implementations, hence for evaluation purposes the tests are performed communicating from a smartphone to a laptop, using both the CoAP and CoAPS protocols, since the second involves the implemented DTLS protocol, allowing to measure its performances as well.

The environment setup is composed of a smartphone (Google Nexus S) and a laptop. The smartphone communicates with the laptop using the CoAP or CoAPS protocol using the wireless (802.11 N) acting as client or as server.

Performances are evaluated using an open source tool called Perf4j [1] that allows to calculate and display performance statistics for the java code. Thanks to this tool it is possible to measure the exact time needed for an operation to be executed, or the times between a request is sent and the response is received.

5.2 Evaluation Criteria

The first phase of the evaluation is the selection of the evaluation criteria. It is necessary to decide what should be tested and the reason this test should be performed. In other words the tests are identified by the question: *"what should I test?"* and validated with the question: *"why should I test this?"*.

Since the purpose of this document is to provide security for the CoAP protocol, and this has been accomplished thanks to the DTLS protocol, in order to understand *what should be tested* it is good to compare the two protocols (CoAP and CoAPS) and analyse the differences.

[1] http://perf4j.codehaus.org/

Let us assume that a CoAP client sends a request to a server and receives the response. In this case the application asks the CoAP client to prepare a request, send it (through UDP protocol) to the server. The server receives the request, elaborates it then generates and send a response that will be delivered to the client as shown in figure 5.1.

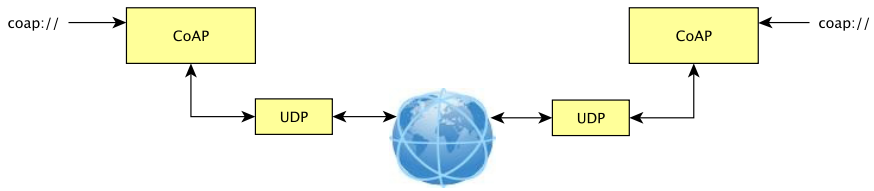

Figure 5.1: CoAP request and response

For the same scenario, using CoAPS the request and response schema is slightly different. The high level representation of CoAPS (figure 5.2) shows that in order to protect CoAP messages there is the need to introduce an intermediate component before sending requests and responses, as explained in detail in section 4 page 26.

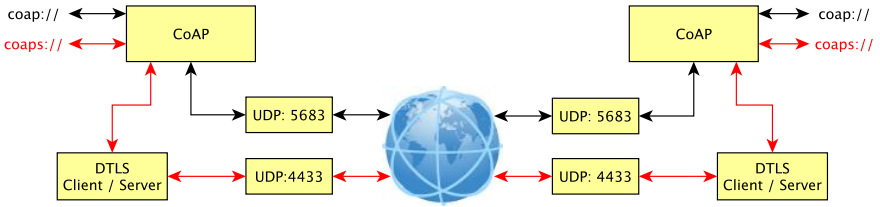

Figure 5.2: CoAPS request and response

Adding intermediate components might introduce delay, so now one of the answers to the question: *"what should be tested?"* is now: the difference (in time) of performing exactly the same operation, using CoAP and CoAPS. According to the principle of psychological acceptability, security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present [2]. The reason (replying then to *why this test is important*) is that if it takes too much time to perform the same operation securely, compared to doing the same thing in a non secure way, the first will not be used. This is *why* this test must be performed (replying to the second question); if it takes too much time to retrieve a CoAPS response, in the long run it will be preferred to use the insecure but faster way of using insecure CoAP.

From the section 2.3 page 9 and from the picture 5.2 it is possible to notice that while for the CoAP protocol the preparation and sending time is always similar for the first and consequent requests/responses, for the CoAPS the behaviour is different. In fact during the first request (assuming that we are sending a CoAPS request from a client), the DTLS protocol needs to handshake with

the server (in this example) the information needed to secure the communication. For this reason the very first request and response will be delayed for the time needed for the DTLS handshake. The consequent requests/responses will be probably on the same magnitude of the CoAP protocol, with some overhead due to the DTLS encapsulation.

So there are several aspects that need to be evaluated in this scenario. First it is needed a comparison between the first request with consequent reception of the response using the protocol CoAP and the same measurement using the protocol CoAPS. Then it is possible to analyse the same behaviour repeating the experiment and measuring the second time a request is sent and a response is receive, for both protocols. The second value will be the same for all the consequent messages (unless the server desires a re-handshake). This will be repeated in case the smartphone is a client, and in case the smartphone is a server, since the behaviour might change and we are interested in the performances in smartphones.

The tests previously described take in consideration both the computational time and the transmission time of messages for the recommended security settings [14]. For the CoAPS protocol, discarding the handshake process, it is interesting to understand how the performance of the protocol vary depending on the payload size. This is important because it is possible to evaluate the behaviour with different sizes of payload, but also since cipher suites and protocols are continuously evolving, this allows to understand how it is possible to "secure" the data with the nowadays smartphones. It is then possible to compare the behaviour with the standard settings, and try to foresee what is the overhead in case there is the need to increase the security level.

5.2.1 Round Trip Time

The round trip time is the time that elapses between when the request is sent, and when the response for the same request is received. In other words, is the time that takes to receive a response, once a request is sent. Since it is not of interest for this document to evaluate how much it takes to the CoAP implementation to handle the messages, the round trip time is here the time from the moment the CoAP implementation sends the request (directly via UDP in case of CoAP, or to the DTLS protocol in case of CoAPS) to the moment in which it receives the response (from UDP or from the DTLS protocol).

Assuming a payload of 8 bytes on both request and response, table 5.1 shows the average, minimum, maximum, standard deviation and number of measurements of the round trip time using the protocol CoAP and CoAPS.

Protocol	Avg(ms)	Min (ms)	Max (ms)	Std Dev	Count
CoAP	20.0	14	29	5.2	5
CoAPS	3387.2	3163	4172	393.3	5

Table 5.1: Response time of the first CoAP and CoAPS request

It is possible to notice that there is a significant difference between the two average values. This is due to the handshake time for the DTLS protocol, as it is shown further on in this section. At the end of both times, the request is

received from the CoAP implementation (decrypted if it is a CoAPS response) ready to be processed and sent to the upper application.

The transmission time for the CoAP protocol is divided in the following way:

- time to transfer the datagram from the client to the server
- time spent in the server for processing the request and generating a response
- time to transfer the datagram from the server to the client

In case of CoAPS, the round trip time is divided in the following operations (in case of first transmission, when the DTLS protocol performs a full handshake):

- time to perform the handshake
- time to encrypt encapsulate and serialize the request
- time to transfer the datagram from the client to the server
- time to parse deserialize and decrypt the request
- time spent in the server for processing the request and generating a response
- time to encrypt encapsulate and serialize the response
- time to transfer the datagram from the server to the client
- time to deserialize decrypt and parse the response

While from the second transmission the handshake time (first element) is not present.

From these first measurements, it has been noticed an interesting behaviour: because of the computational power of the smartphone, in order to avoid retransmissions during the handshake, it has been needed to increase the retransmission timer from 1 to 3 ms [2]. In particular the flights 4 and 5 required more than 1 second from the preparing state to the sending state, making the other peer retransmitting the last flight since it did not receive any response within the expiration of the timer. Moreover the server requires more time to prepare and send the flight 4, than the client to send the flight 5 due to the cipher suite. This is a required adjustment to be done on the smartphones; it is not needed on a normal computer, or for a more powerful smartphone or tablet.

It is interesting now to see how much time it is needed in order to perform the same operation for the consequent request, since the CoAPS will have a different behaviour. This is shown in table 5.2 comparing the same CoAP request and the new CoAPS response.

The CoAPS round trip time time is decreased to approximately three times the CoAP round trip time with a minimum value slightly bigger than the CoAP value. The reason of different values for maximum, minimum and standard deviations are explained in section 5.3 page 46.

[2]The RFC **??** recommends a retransmission timer of 1 ms

Protocol	Avg(ms)	Min (ms)	Max (ms)	Std Dev	Count
CoAP	20.0	14	29	5.2	5
CoAPS	69.1	32	156	42	10

Table 5.2: Response time of the second CoAP and CoAPS request

Handshake comparison

In table 5.1 the response time in case of secure CoAP takes more than three seconds. This is due to the DTLS handshake, but how this time is divided during the handshake? The upper half of table 5.3 shows the time needed for a smartphone acting as a client to prepare and send each flight. Having in mind the handshake process (figure 2.4 page 10) it is possible do some considerations. The most time expensive flight is as expected the flight 5 since it involves asymmetric cryptography (used for the elliptic curves Diffie-Hellman in the *ClientKeyExchange* and for signing in *CertificateVerify*). The least time consuming is instead the flight number three, since it is just a retransmission of the first message with an additional value. The first flight instead is more computationally expensive because of the generation of a 256bit random number (and in order to have good entropy some are needed several cycles of seeding). The sum of all the single flights is not the total handshake time since those numbers are the only sum of the time needed by the DTLS client to prepare and send the flights. This time doesn't include the transmission time and the computations needed on the server side before sending the next flight. During the handshake in fact the transmission is synchronous; the next flight cannot be sent unless all the previous and required flights are received).

Protocol	Avg(ms)	Min (ms)	Max (ms)	Std Dev	Count
Client Handshake	3387.2	3163	4172	393.3	5
Flight 1	20.6	14	30	6.0	5
Flight 3	1.0	1	1	0	5
Flight 5	300.4	269	329	24.6	5
Server Handshake	4881.4	4467	5789	489.6	5
Flight 2	2.4	1	6	2.1	5
Flight 4	698.6	656	750	40.3	5
Flight 6	3.2	2	5	1.2	5

Table 5.3: DTLS Handshake

Measuring the handshake time needed for a DTLS server located on a smartphone, inverting then the previous measurement (second half of table 5.3), it is easy to notice that the handshake times are not the same. So performing the handshake as a client on a smartphone it does not take the same amount of time than performing the handshake on a smartphone used as server. Analysing the most expensive flight (flight 4) that involves asymmetric cryptography it is possible to see that takes more than double the time for the most expensive flight for a DTLS client. This is due to the fact that in the server flight 4 the operations are more computationally expensive than the client flight 5 such as chose a point in the previously negotiated elliptic curve and use it to generate a key pair, in order to perform the ephemeral elliptic curve diffie hellman for the

key exchange (explained in section 2.8 page 15).

Examining the values of the total handshake the server needs more time to perform the full handshake reaching almost double the time needed for the client handshake in the worst scenario, in which for client handshake it is meant the time needed to perform a full handshake running the DTLS protocol on a smartphone acting as a client, and the opposite statement for the server handshake.

Now that it is clear the time that a client needs to wait from the moment it sends a request, to the moment it receives a response, for both CoAP and CoAPS protocols, it is possible to proceed analysing the round trip time for consequent requests, since as shown above, the time for secure CoAP is comparable with the time needed to use the insecure protocol.

Keeping the same setup and varying only the number of bytes of payloads in the request and in the response, it is possible to compare the performances with and without security of the Constrained Application Protocol. In figure 5.3 is shown the round trip time needed for the two protocols, with the same payload in both requests and responses, from 0 (only CoAP header) till to 512 bytes.

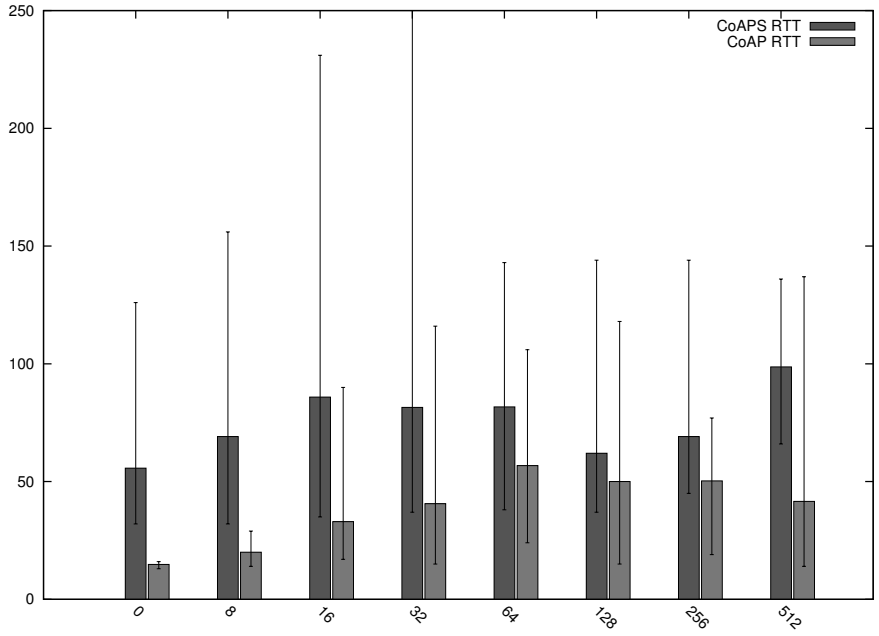

Figure 5.3: Round trip time comparison of CoAP and CoAPS

Since the CoAP header needs 16 bytes, the first measurement (with 0 bytes of payload in both request and response) is the minimum round trip time. In this case there is a substantial difference between CoAP and CoAPS, due to all the additional operations that are needed to secure the communication. Increasing the payload, it is interesting to notice that for both CoAP and CoAPS after 64 bytes of payload, there is a reduction of the round trip time. This is more linear for the CoAP protocol than for CoAPS. This behaviour is probably

due to the underneath implementation and handling of network, caching and other operations from Android. Moreover, as explained in section 5.3 these measurements are susceptible to two different phenomena: cpu fluctuations and transmission fluctuations. With a payload from 64 to 256 bytes, the difference in round trip time between CoAP and CoAPS is minimal, resulting in having a more secure communication with almost non influent performances, a part for the initial mandatory handshake time.

5.2.2 CPU Time Overhead

All the previous measurements have been done using the default and suggested cipher suite in CoAP [14] and explained in section **??**. Since smartphone's performances are always increasing, as well as the performances of other computers, it makes sense to think that in the next future it will be needed more strength to protect the communication. This is due to the fact that the ciphers can be deciphered if the attacker possesses a high computational power, because it reduces the time needed for brute-force, educated guess, and so forth.

Since for the asymmetric encryption, are already used elliptic curves, granting a higher level of protection compared to the traditional factoring problem with the sam key size and they are used only during the handshake, the focus will here be on symmetric encryption.

Symmetric encryption and decryption, as explained in section 2.9 16, are responsible (in case of CoAPS in the DTLS protocol) for the encryption and decryption of the CoAP messages. Every message, after the handshake is completed, is encrypted using symmetric encryption (since less computationally expensive). In this scenario it is evaluated the proposed CoAP encryption [14]: AES with 128 a key of 128 bits in CCM mode [16]. The idea is that, for the reasons above explained, it might be needed to increase the key size, and the AES algorithm with 256 bits of key in the same mode is already available, so it is possible to compare how much additional computation is needed in order to use a bigger key size (and consequently having stronger communication security).

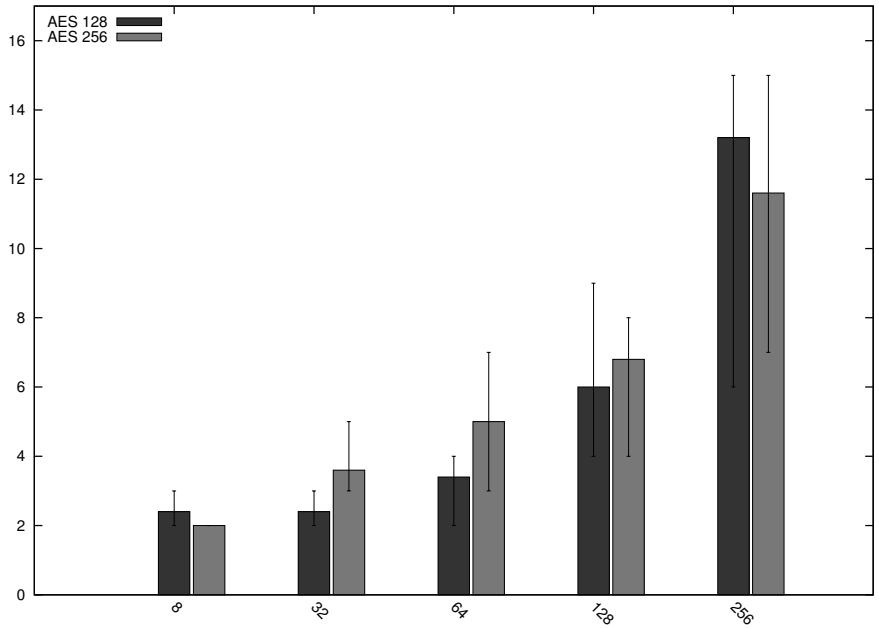

Figure 5.4: Encryption comparison between AES 128 and AES 256

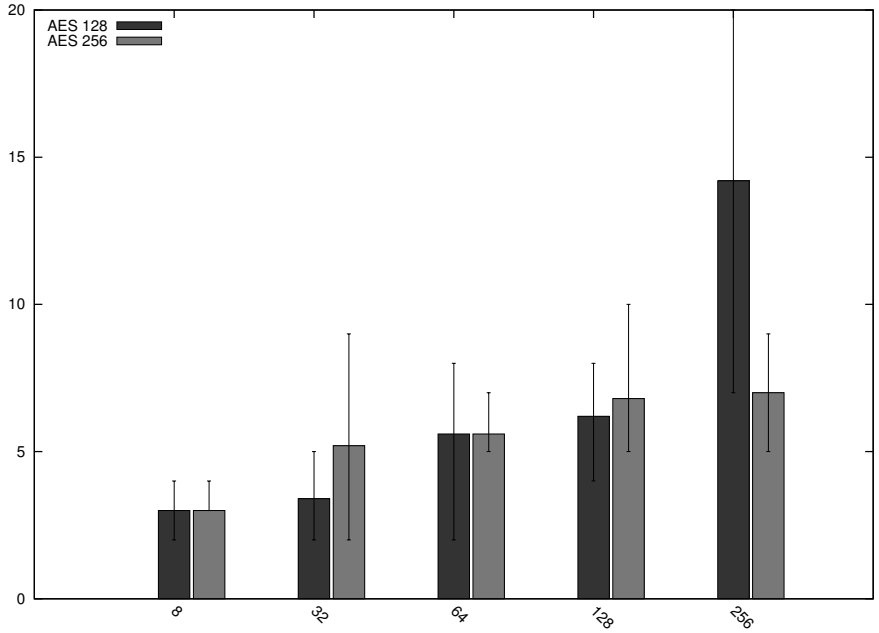

Figure 5.5: Decryption comparison between AES 128 and AES 256

Figures 5.4 and 5.5 shows the processing time in order to encrypt (figure 5.4) and decrypt (figure 5.5) five different payloads with the AES encryption

algorithms with 128 and 256 bits composing the key. To this measurements are always added the 16 bytes of CoAP header, and the operation is performed in a smartphone.

It is possible to notice that the computational time it is not substantially different, allowing a more secure communication at the expense of some milliseconds of computational time (that should be added on both request and response). As explained in the next section, there are some fluctuations in the measurements, due to the fact that the smartphone's cpu is being used by other applications running in background, and even though the standard deviation is lower than in case of wireless transmission, for small measurements this affects the results.

5.3 Accuracy of the measurements

As previously stated, data are collected and elaborated thanks to a java library called perf4j. The measurements are done in two steps: first the data is collected, injecting the "start" and "stop" directly in the source code in order to measure precisely the desired functionality. This is essentially done adding a tag for each measurement, that is consequently written in a text file. The same library is then able to parse and identify different tags and corresponding timestamps in order to calculate average, minimum and maximum values, standard deviation and number of time the measurement has been recorded. This library is able to measure with an accuracy of milliseconds. So for measuring small time intervals this framework might not be precise enough. Moreover the time values of the measurements are output as a log, making easy to obtain these information from a smartphone, since all the measurements have been done in a smartphone.

On all the measurements that involve the quantification of a certain time, and a part of this time is a datagram sent o received over the wireless network, it is present a fluctuation. Due to the highly variations in wireless transmission (i.e. packet loss) measuring the time needed for a packet to be transmitted over the air gives different results in different times. If this measurement is repeated for several times in the same evaluation, the sum of all these variations might result in a measurable variable error. It is possible to notice this behaviour in table 5.1. In fact the standard deviation, or in other words, the different between the measurements and their average, is almost half second, for a measurement that in the average takes slightly more than three seconds. In this case this transmission fluctuation due to the transmission over the wireless is present several times. During the handshake, in fact, there are several messages exchanged, and each brings its fluctuation in the total sum.

Another type of fluctuation, more obvious in section 5.2.2 is the CPU fluctuation. On the android phone (as well as other smartphones) there are several applications running in background, monitoring or doing other operations. It is impossible to close most of them, resulting in a 5% variation of the CPU load. When performing precise measurements like the encryption or decryption time, in which the CPU is directly involved in the measurements, this creates a discrepancy between the same measure in different times. This is the reason some graph, for instance figure 5.5 in case of 256 bytes of payload, the decryption with AES 128 takes more time than the encryption using AES 256 (observable thanks to the maximum and minimum value reported in the graphs). This error

is present in every measurement, but it can be neglected when the transmission fluctuation is present (or in other words when measurements involve measuring time and there is involved transmission over wireless network).

Chapter 6

Conclusions

The objective of this document is to provide confidentiality and integrity for machine to machine communications between Android devices and other endpoints such as sensor nodes in the Internet of Things through the CoAPS protocol. This can give innumerable benefits.

Nowadays smartphones and tablets are every day's accessory, and probably soon an essential tool. The whole world is becoming more and more technological, with the possibility to interconnect all these devices through the Internet. The security plays a big role in this scenario, in order to prevent intruder from tampering private accessories. With embedded devices is not always feasible and convenient to use protocol such as TCP (protected by the widespread TLS protocol), because of its design. More and more devices and applications use the faster UDP protocol (that needs though particular care for the data transmission). This protocol can be secured with the implemented DTLS protocol, allowing, for instance, to keep under control the house fridge or freezer, or the oven, and so on from a smartphone or a tablet, in a secure way.

This has been accomplished through the implementation of a DTLS API for Android. The implementation allows an application to send and receive encrypted data using UDP datagrams as transport. This API is used in one of the CoAP implementation, Caifornium, in order to allow secure CoAP messages exchange between two peers. It is also possible to use this API in other applications; when there is the need to secure data, the application will use the DTLS API instead of using directly the UDP protocol, on both sender and receiver. The result is the interoperability between Android devices, sensors in the Internet of Things, and other CoAPS-enabled devices, which before this work, was not possible in a secure way, also suitable for constrained environments.

Chapter 7

Future Work

A basic requirement in order to allow the use of CoAPS between a smartphone and a sensor in the IoT, it is needed to implement the DTLS protocol on a sensor network. In particular an operating system such as Contiki can be used to host the implementation, so that the sensor will host a DTLS server, allowing a smartphone to retrieve data (for instance temperature, humidity, etc in the sensors of the sensor node) securely over the internet. This is the objective of this thesis, but in order to allow the full communication, the work must be done both in smartphones and sensor networks (or other environments).

To fulfil the objective of this thesis it has been required to implement several features, most of which were not existing, leading to more than 5000 lines of java code. Because of this, it has been implemented what was necessary to fulfil the goal given the time constraint. There is much work to do to enhance this protocol; the code will be released as a patch for the Bouncy Castle security provider, and then repackaged in Spongy Castle, so any application that wants to use this protocol, will just need to include this jar API in the application, and use the methods exposed in the interface. This allows not only android phones to use this particular DTLS cipher suite, but also any other java applications.

Regarding the implementation there are several parts to be completed in order to have a full and stable protocol that has not been prioritised in this context since the objective is to offer a proof of concept for the secure CoAP.

The DTLS protocol allows fragmentation for handshake messages, in case the MTU is smaller than the dimension of the datagram (and this might happen when sending a long certificate chain), but since CoAP messages are notoriously small, and the certificate chain is smaller than the MTU the fragmentation was omitted.

Session resumption is also important to be implemented, since it will allow a re-handshake that will be performed in less time than the full handshake (and as shown in section 5 the full handshake time might take several seconds). The server *HelloRequest* message has to be implemented as well to allow a resumed handshake, as well as a system of session resumption, to keep track the session state of each client.

In order to extend the compatibility is also required to implement additional cipher suites and support for elliptic curves (fortunately most of the work can be done using the already implemented TLS cipher suites in the Bouncy Castle).

Regarding the evaluation, it is interesting to evaluate the same performances

with other different kind of smartphones in order to effectively determine the overhead based on the mobile phone used. On different platforms, different cipher suites and security levels can be evaluated according to the needs. Moreover, smartphones become more powerful everyday, as well as stationary computer of potential attackers; in the future it might be also needed to introduce new cipher suites in order to maintain a high protection. Performances on other devices such as tablets are also important, since they are more and more used and widespread.

Bibliography

[1] C. Adams and S. Farrell. Internet X.509 Public Key Infrastructure Certificate Management Protocols. RFC 2510 (Proposed Standard), March 1999. URL `http://www.ietf.org/rfc/rfc2510.txt`. Obsoleted by RFC 4210.

[2] Matt Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 1 edition, November 2004. ISBN 0321247442.

[3] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. URL `http://www.ietf.org/rfc/rfc4492.txt`. Updated by RFC 5246.

[4] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. URL `http://www.ietf.org/rfc/rfc5280.txt`.

[5] S. Deering and R. Hinden. *RFC 2460 Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, December 1998. URL `http://tools.ietf.org/html/rfc2460`.

[6] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. Internet Engineering Task Force, August 2008. URL `http://www.ietf.org/rfc/rfc5246.txt`. Updated by RFCs 5746, 5878, 6176.

[7] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. URL `http://www.ietf.org/rfc/rfc2409.txt`. Obsoleted by RFC 4306, updated by RFC 4109.

[8] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network security: private communication in a public world, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2002. ISBN 9780137155880.

[9] D. McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116 (Proposed Standard), January 2008. URL `http://www.ietf.org/rfc/rfc5116.txt`.

[10] D. McGrew, D. Bailey, M. Campagna, and R. Dugal. Aes-ccm ecc cipher suites for tls, November 1 2012.

[11] D. McGrew, K. Igoe, and M. Salter. Fundamental Elliptic Curve Cryptography Algorithms. RFC 6090 (Informational), February 2011. URL http://www.ietf.org/rfc/rfc6090.txt.

[12] Nagendra Modadugu and Eric Rescorla. The design and implementation of datagram TLS. *IN PROC. NDSS*, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.8758.

[13] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. URL http://www.ietf.org/rfc/rfc6347.txt.

[14] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained application protocol (coap), December 2012.

[15] Jean-Baptiste Waldner. *Nanocomputers and swarm intelligence*. ISTE ;;John Wiley, London ;Hoboken NJ, 2008. ISBN 9781848210097.

[16] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610 (Informational), September 2003. URL http://www.ietf.org/rfc/rfc3610.txt.

Appendix A

How to run the application

Hardware Prerequisites: The followings are the items needed to run the application:

- A computer with wireless card
- An Android phone with wireless card
- A USB cable to connect the Android phone to the computer

Software Prerequisites: The following is the software needed in order to build the application in the computer and transfer it to the smartphone:

- A Java Virtual Machine
- Eclipse IDE
- Android SDK with Eclipse Plugin

Building the application: The default settings for the application are the use of the cipher suite: TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 and mutual authentication, hence before running the application there is the need to build the certificates, to be used in the desktop application and in the Android application.

1. Install the Java Virtual Machine, Eclipse and the Android SDK
2. Import the sources of the CoAPS application in the eclipse workspace
3. Generate the certificate hierarchy executing the class SCCertGenerator in the package org.spongycastle.crypto.dtls as a Java application. This will generate the certificates needed to run the application, giving as output the name and description of the certificates. They will be placed in the default location in order to be used by the CoAPS protocol. These files (client.p12, server.jks, server.p12 and trustStore.jks) must be copied in the smartphone
4. copy the above mentioned files in the smartphone in the folder "certs" in the SD card of the smartphone

Now it is possible to execute the whole package as Android application. Eclipse will build and install the application in the Android phone. This operation is necessary only to install the application the first time. Consecutive executions will not need the USB connection between the Android phone and the computer.

Once the proper IP addresses are specified, it is possible to start the Server on the computer, and run the client from the smartphone, or vice versa.