ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RESEARCH GROUP FOR
Distributed Systems

Master's Thesis

at the Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

Securing the Constrained Application Protocol

by Stefan Jucker

Autumn 2012

ETH student ID: 06-912-414
E-mail address: sjucker@student.ethz.ch

Supervisors: Dipl.-Ing. Matthias Kovatsch
Prof. Dr. Friedemann Mattern

Date of submission: 10 October 2012

Declaration of Originality

I hereby declare that this written work I have submitted is original work which I alone have authored and which is written in my own words.
With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette:'

`http://www.ethz.ch/students/exams/plagiarism_s_en.pdf`

The citation conventions usual to the discipline in question here have been respected.
This written work may be tested electronically for plagiarism.

Zurich, 10 October 2012

Stefan Jucker

Acknowledgments

I would like to thank Matthias Kovatsch for the support and guidance during the process of writing this master's thesis. My gratitude also goes to my interoperability test partners, particularly Daniele Trabalza with whom I have spent many exhausting hours of debugging, fixing and testing of our implementations.

Finally, I would like to thank my family and my friends, especially Ueli Ehrbar, for their support throughout my studies and the master's thesis.

Abstract

Although the security needs for the Internet of Things are well-recognised, it is still not fully clear how existing IP-based security protocols can be applied to this new environment with constrained nodes and low-power, lossy networks. This thesis provides a state-of-the-art survey and a roadmap specifying the necessary steps to realise the security suite for different Constrained Application Protocol (CoAP) implementations of the Distributed Systems Group. As a first step, we implemented Datagram Transport Layer Security (DTLS) for Californium (Cf), a CoAP framework written in Java for unconstrained environments. This implementation provides all the specified mandatory functionality and has been tested for interoperability with three other DTLS implementations. Analysed in an unconstrained environment, a full handshake increases the round-trip time by at least 130ms compared to an unsecured CoAP request, which takes about 40ms. Based on the performance and profiling analysis, and the gained experience during the implementation phase, we present an outlook on the future work that needs to be undertaken towards a constrained environment.

Contents

Glossary

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

CA Certificate Authority

CCM Counter with CBC-MAC

Cf Californium

CoAP Constrained Application Protocol

CoRE Constrained RESTful Environments

CPU Central Processing Unit

CSR Certificate Signing Request

Cu Copper

DANE DNS-based Authentication of Named Entities

DER Distinguished Encoding Rules

DoS Denial-of-Service

DSA Digital Signature Algorithm

DTLS Datagram Transport Layer Security

ECC Elliptic Curve Cryptography

ECDH Elliptic Curve Diffie-Hellman

ECDSA Elliptic Curve Digital Signature Algorithm

Er Erbium

ESP Encapsulating Security Payload

HMAC Hash-Based Message Authentication Code

IANA Internet Assigned Numbers Authority

IETF Internet Engineering Task Force

IoT Internet of Things

IP Internet Protocol

IPsec Internet Protocol Security

JAR Java Archive

JCA Java Cryptography Architecture

JDK Java Development Kit

JVM Java Virtual Machine

MAC Message Authentication Code

MCU Microcontroller Unit

MTU Maximum Transmission Unit

PFS Perfect Forward Secrecy

PRF Pseudorandom Function

PSK Pre-Shared Key

SECG Standards for Efficient Cryptography Group

SICS Swedish Institute of Computer Science

TLS Transport Layer Security

TLV Type-Length-Value

UDP User Datagram Protocol

Objectives

The student will assess the drafts in the *Constrained RESTful Environments (CoRE)* working group as well as available implementations of the underlying security mechanisms and discuss possible threats to the protocol and its limitations. Based on these results, a roadmap will be created listing the steps that need to be taken to realise the security suite for the *CoAP* implementations of the Distributed Systems Group: *Californium (Cf)*, *Erbium (Er)*, and *Copper (Cu)*. The theoretical results will be applied to *Cf* by implementing all mandatory security options of the latest *CoAP* Internet-Draft. In a final step, the student shall evaluate the realised security suite qualitatively through threat modelling and quantitatively through performance measurements.

1 Introduction

1.1 Motivation

The *Internet Engineering Task Force (IETF)* is currently standardising the *Constrained Application Protocol (CoAP)* [37] and with this the need for standardised security follows naturally. In the Internet, the de facto standard to achieve security is *Transport Layer Security (TLS)* paired with a public-key infrastructure. However, this protocol is not suited for constrained networks due to its need for reliable messaging and also its high demand on the hardware. Therefore, multiple alternatives have been proposed to solve the security requirements for constrained environments.

The *CoAP* specification suggests either the use of *Datagram Transport Layer Security (DTLS)* or *Internet Protocol Security (IPsec)* to achieve data origin authentication, integrity and replay protection, and encryption for *CoAP* messages [37]. Additionally, other protocols and modifications of the mentioned approaches exist which are undergoing review by the *IETF* and *Constrained RESTful Environments (CoRE)* working groups. So far, however, no standard for end-to-end security could be established. This is especially difficult due to the discrepancy between the demand for high security standards and the available envisioned inexpensive but constrained hardware.

Before network security can be applied, the small devices, mostly without any user interface, must be commissioned into the network. This *security bootstrapping* poses also a challenge that needs to be solved for an effective use of *CoAP*. The same is true for the physical vulnerabilities of devices: They could easily be stolen or forcibly altered to compromise security.

1.2 Contribution

By the start of this master's thesis, no implementation existed that fully conformed to the *DTLS* specification. This thesis presents a *DTLS* implementation, integrated into the *CoAP* framework *Californium (Cf)*, that conforms to all mandatory requirements defined in the *DTLS* [33] and *CoAP* [37] specifications. Furthermore, it gives an in-depth analysis of the different security approaches and presents ideas for optimising security specifications and the current implementation to be more friendly towards constrained environments.

1.3 Outline

After introducing the *Constrained Application Protocol (CoAP)*, the thesis provides a state-of-the-art survey in Chapter 3 about the current security measures and outlines a roadmap specifying the necessary steps to achieve a secure *CoAP* implementation. Chapters 4 and 5 document the implementation for *CoAP* over *DTLS* deployed in an unconstrained environment. The following Chapter 6 analyses the implementation according to its security and performance. The remaining chapters will present the achieved results, discuss them and draw the conclusion.

2 The Constrained Application Protocol

2.1 Overview

The *Constrained Application Protocol (CoAP)* is a specialised Web transfer protocol for use with constrained nodes (e.g., low-power sensors, switches, or valves) and constrained (e.g., low-power, lossy) networks. The *Constrained RESTful Environments (CoRE)* working group aims at realising the REST architecture in a suitable form for the most constrained nodes and networks [37]. One design goal of *CoAP* has been to keep message overhead small, thus limiting the use of fragmentation and sparing constrained networks, like 6LoWPAN, to execute expensive fragmentation of IPv6 packets. *CoAP* is an application layer protocol that easily translates to HTTP for integration with the existing Web while meeting specialised requirements such as multicast support, very low overhead and simplicity for constrained environments, and machine-to-machine applications (e.g., [8], [23]). It makes use of two message types, *requests* and *responses*, using a simple binary fixed-size header format which is potentially followed by options in *Type-Length-Value (TLV)* format and a payload. The length of the payload is determined by the datagram length and when bound to the *User Datagram Protocol (UDP)* (the standard case), it must fit within a single datagram.

CoAP has the following main features [37]:

- Constrained Web protocol fulfilling machine-to-machine requirements.
- UDP binding with optional reliability supporting unicast and multicast requests.
- Low header overhead and parsing complexity.
- Push notifications through a publish/subscribe mechanism.
- Simple proxy and caching capabilities.
- Security binding to *Datagram Transport Layer Security (DTLS)*.

One of *CoAP*'s most promising applications will be the *Internet of Things (IoT)*: At the beginning of the *IoT*, everyday objects were interconnected through their virtual representation using barcodes and later RFID. Now, the IP-based *IoT* becomes reality with tiny embedded devices that are directly connected to the Internet over IP [24], enabled by *CoAP* which completes the IP stack (in combination with 6LoWPAN [19] and RPL [43]) for such constrained devices.

2.2 CoAP Implementations

Currently, there exist three implementations of *CoAP* developed at the Institute for Pervasive Computing, ETH Zurich: *Californium (Cf)*, *Erbium (Er)*, and *Copper (Cu)*.

2.2.1 Californium

Cf[1] is a *CoAP* framework written in Java and developed for the use in unconstrained environments. *Cf* is based on a layered architecture, extending the two-layer approach described in the *CoAP* draft [37]. This allows an isolated implementation of different aspects such as *message retransmission*, *transactions*, *block-wise transfer* and of course *security* as we will discuss later in detail. The architectural design of *Cf* is documented in the lab report [29] of the initial implementation. Please note that due to the continuous advancement of *Cf*, some described design decisions in the lab report are already outdated.

2.2.2 Erbium

Er is a low-power REST Engine for the Contiki operating system[2] which allows tiny, battery-operated, low-power systems to communicate with the Internet [21]. Therefore, this implementation is specialised for constrained environments as it is designed to run on small amounts of memory and low-power *Central Processing Units (CPUs)* or *Microcontroller Units (MCUs)*.

2.2.3 Copper

Cu[3] is a *CoAP* user-agent for Firefox implemented in JavaScript. It enables users to browse *IoT* devices in the same fashion they are used to explore the Web. This is done by providing a presentation layer that is originally missing in the *CoAP* protocol suite. Since it is embedded into Firefox as an add-on, it is designed for unconstrained environments as *Cf*. Its ability to render a number of different content types such as JSON or the *CoRE* Link Format makes it a useful testing tool for application as well as protocol development [24].

[1] https://github.com/mkovatsc/Californium
[2] http://www.contiki-os.org/
[3] https://github.com/mkovatsc/Copper

2.2.4 Others

Lerche, Hartke and Kovatsch list in their *CoAP* survey additional *CoAP* implementations [24]:

jCoAP

jCoAP[4] is a Java implementation for unconstrained devices and embedded systems such as Java-based smartphones and mobile devices (e.g., Android). It provides a *CoAP*-to-HTTP and HTTP-to-*CoAP* proxy which performs protocol translations between the two protocols.

libcoap

libcoap[5] is a library for *CoAP* message parsing, serialisation, and transmission. It has been ported to different embedded system architectures, in particular the operating systems Contiki and TinyOS[6].

[4] http://code.google.com/p/jcoap/
[5] http://libcoap.sourceforge.net/
[6] http://www.tinyos.net/

3 Security Survey

3.1 Security Considerations

This section gives an overview about the security threats and vulnerabilities in the *IoT* and should sharpen the focus on what must be considered when trying to secure *CoAP*. Garcia-Morchon, et al. identify nine potential security threats, which are summarised in Table 3.1 ordered by the vulnerable layer and the lifetime phase of a thing [13].

	Manufacturing	**Installation, Commissioning**	**Operation**
Physical Thing	Device Cloning	Substitution	Privacy Threat Extraction of Params
Application Layer			Firmware Replacement
Transport Layer		Eavesdropping Man-in-the-Middle	Eavesdropping Man-in-the-Middle
Network Layer		Eavesdropping Man-in-the-Middle	DoS Routing Attack
Physical Layer			DoS

Table 3.1: The security threat analysis. The potential attacks are grouped by the lifetime phase and the potential point of vulnerability.

Roughly, the security threats can be divided into two categories: *physical* (local) attacks and *non-physical* (remote) attacks. Physical attacks are executed by attackers which force their way into the physically unprotected thing and try to compromise it in different ways [5, 13].

Cloning of Things
An untrusted manufacturer can easily clone the physical characteristics, firmware and software, or security configurations during the manufacturing process of a thing. The manufacturer could sell the cloned thing at a cheaper price in the market to gain financial benefits or even worse implement additional functionality which could later be used with malicious intent.

Malicious Substitution of Things
During the installation of a thing, a genuine thing may be replaced with a low-quality variant without being detected. The main motivation may be cost savings by significantly reducing the installation and operational costs.

Firmware Replacement Attack
After a certain time into the operational phase, the software or firmware of a thing may be updated to introduce new functionality or new features. During this maintenance phase, an attacker may be able to exploit such an upgrade by replacing it with malicious software. If such an attack is successful, the operational behaviour of this thing can be influenced and exploited afterwards.

Extraction of Security Parameters
Things deployed in ambient environments are normally physically unprotected and can easily be captured by an attacker. When in possession of a thing, the attacker may try to extract certain security information such as keys from this thing. Once a key is compromised (e.g., a group key), the whole network may be compromised as well.

Denial-of-Service Attack
A *Denial-of-Service (DoS)* attack can be launched by physically jamming the communication channel.

All of the described attacks depend on a physically present attacker. Therefore, the security measures that need to be taken to protect things against such attacks go beyond the scope of Internet protocol security. As we focus on securing *CoAP*, the countermeasures against these attacks will only be discussed in combination with Internet protocol security.

The non-physical (remote) attacks include the following [3, 13, 28]:

Eavesdropping Attack
During commissioning of a thing into a network, it can be vulnerable to eavesdropping: If security parameters or keying material are exchanged in the clear, the attacker might be able to retrieve the secret keys established between the communicating entities. But even a protected network can be vulnerable to eavesdropping, e.g., in the event of session key compromise which can happen after a longer period of using the same key without renewing it.

Man-in-the-Middle Attack
A man-in-the-middle attack is possible during the commissioning phase, e.g., when keying material is exchanged in the clear and the security of the key agreement algorithm depends on the assumption that no third party is able to eavesdrop on or sit in between the two communicating entities. During the operational phase, man-in-the-middle attacks are only possible when anonymous connections are used.

Routing Attack
Routing attacks include amongst others a) *Sinkhole attack*, where the attacker declares himself to have a high-quality route, allowing him to do anything to all packets passing through. b) *Selective forwarding*, where the attacker may drop or forward packets selectively.

Denial-of-Service Attack
An attacker can continuously send requests to be processed by specific things and therefore exhaust their resources. This is especially dangerous in the *IoT* since things usually have tight memory and limited computation capabilities.

Privacy Threat
The tracking of the location and the usage of a thing may pose a privacy risk to its users. An attacker could deduce behavioural patterns of the user by inferring information based on the gathered data.

As we have identified potential security threats, let us define what we want to achieve when we talk about *security requirements* against these threats [14, 28, 41]:

Confidentiality
Confidentiality describes the prevention of disclosure of information to unauthorised entities. We want to achieve confidentiality to prohibit *privacy threat* and *eavesdropping* attacks. Please note that an attacker can observe communication patterns of a user even if confidentiality is provided by the connection, allowing him to infer private information about the user anyway. This attack is just complicated when confidentiality is provided, but not fully averted.

Authenticity
Authenticity guarantees that all parties involved in the communication are who they claim they are.

Integrity
Integrity is violated if a message can actively be altered during transmission without being detected. If message integrity and authenticity is guaranteed, *man-in-the-middle* attacks can be averted.

Non-Repudiation
Non-repudiation implies that one party of a transaction cannot deny having received a transaction nor can the other party deny having sent a transaction.

Availability
Ensuring the survivability of services to parties when needed, even during a *DoS* attack.

Authorisation
Authorisation is the function of specifying access rights to resources.

Data Freshness
This can mean data freshness as well as key freshness. It ensures that no adversary can replay old messages.

These security services can be implemented by a combination of cryptographic mechanisms such as block ciphers, hash functions, or signature algorithms and non-cryptographic mechanisms, which implement authorisation and other security policy enforcement aspects.

So far, the listed security threats and goals can be applied to arbitrary networks. We, however, are focusing on constrained networks, therefore we need to look at the additional consequences that arise in constrained environments. One additional problem is the small packet size, which may result in fragmentation of larger packets in security protocols (e.g., a large key exchange message). This may open new attack vectors for state exhaustion *DoS* attacks [13]. Further, the size and number of messages should be minimised to reduce memory requirements and optimise bandwidth usage, while maintaining high security standards. When reducing or simplifying a security protocol in order to minimise energy consumption, one must also expect losses in the security quality [30]. An appropriate trade-off must be found for each individual environment. Another problem is the still existing gap between Internet protocols and the *IoT*, namely *6LoWPAN* and *CoAP*, due to performance reasons. These differences can easily be bridged using protocol translators at gateways, but it becomes a major obstacle once end-to-end security measures between *IoT* devices and Internet hosts are used. When a message is protected using message authentication codes or encryption or both, the protected parts of the message become immutable. Thus, making rewriting not possible for the translators [13].

As can be seen in Table 3.1, there are roughly three different phases in the lifetime of each individual thing, each phase exposed to different threats and having particular security requirements. We will go through these three phases and describe the threats and available techniques to protect a thing against those.

3.2 Manufacturing Security

The life of a thing starts when it is manufactured. There are several difficulties during this phase: a) Normally, there is no single manufacturer that creates all nodes which must interact with each other. Therefore, interoperability and trust bootstrapping between

nodes of different vendors is important. b) An untrusted manufacturer can easily clone the physical characteristics, firmware, or security configuration of a thing and sell it for a cheaper price, or harm the original manufacturer's reputation due to lower quality standards. c) A manufacturer can implement additional functionality within the cloned thing, such as a backdoor [13].

To solve these security problems, no classic cryptographic mechanism can be used. Rather, legal methods must be applied, like having contracts with the manufacturer regulating these issues. Of course, this technique does not result in tight security, but during this phase total security cannot be achieved. One needs to hope that trust can be established on pain of penalties.

3.3 Security Bootstrapping

As can be seen in Table 3.1, there are several security threats during the commissioning phase. In this phase, a device is first installed and then provided with an identity, secret keys, and a list of identities of nodes it can communicate with during normal operation. Thus, it is crucial that the commissioning phase is not compromised, e.g., an attacker could eavesdrop on a unsecured connection and extract operational keying material. That is why we need security bootstrapping before the network can operate normally. Bootstrapping is complete when settings have been securely transferred prior to normal operation in the network.

Sarikaya, et al. describe a framework for initially configuring smart objects securely [35]. They name two distinct phases during the bootstrapping process: the *provisioning* and *bootstrapping* phase. In the provisioning phase, the thing is provisioned with statically configured parameters like certificates (potentially, this has already been done during the manufacturing phase), which are needed in the bootstrapping phase. Using this statically configured information, the dynamically configured information is set up. Further, they define two distinct channels through which two nodes can communicate: the *control* and *data* channel. The control channel is used during the bootstrapping phase and the data channel during normal network operation. When the control and data channel are identical, the bootstrapping is done *In-Band*, otherwise *Out-of-Band*. Depending on whether the control channel is secured or not, different bootstrap security methods can be chosen. Typically, some high-security method should be used to generate a shared secret, which is then used to secure the actual bootstrapping channel using symmetric encryption. The authors propose several methods and note that the negotiated technique should take advantage of the available hardware resource (such as hardware encryption accelerators) [36]:

None
No encryption or authentication is provided, the messages are exchanged in cleartext. It is assumed that some other layer provides security.

Asymmetric with User Authentication, Followed by Symmetric
A shared secret is generated using a Diffie-Hellman style key exchange. The authentication will be provided by the *user*, by confirming cryptographic signatures between two devices. With the shared secret, some symmetric encryption is used to secure the actual bootstrapping channel.

Asymmetric with Certificate Authority, Followed by Symmetric
Similar as the previous method, but instead of required user interaction, a certificate authority provides authenticity.

Cryptographically Generated Address Based Address Ownership Verification
This method includes binding a public signature key to an IPv6 address which can then be used to verify the address ownership. Messages sent from this IPv6 address can be protected by attaching the said public key and signing the message with the corresponding private key. The advantage about this approach is that the protection works without a *Certificate Authority (CA)* or any security infrastructure [4].

3.4 Operational Security

The operational phase starts after the bootstrapping phase is completed. In this phase, *CoAP* is the main transfer protocol used for communication between things. The *CoAP* specification defines two ways to provide security and authentication during the operational phase: *DTLS* and *IPsec*. They differ both at their application as well as their resulting security.

3.4.1 Internet Protocol Security

The *IPsec Encapsulating Security Payload (ESP)* is a protocol for securing *Internet Protocol (IP)* communications by encrypting and authenticating each *IP* packet [20]. Unlike *DTLS*, it operates at the *network layer* of the Internet protocol suite and therefore it is transparent to the application layer and does not require any considerations for a *CoAP* implementation. This fact has its upside as well as its downside: It provides confidentiality and data-origin authentication at the price of approximately 10 bytes per packet [37] without requiring the *CoAP* implementation to be altered, but it may be not appropriate for many environments due to unavailable support in the *IP* stack or missing access to configure it.

3.4.2 DTLS

DTLS is a protocol for securing network traffic which has been specified by the *IETF* in RFC 6347 [33]. Unlike its role model *TLS*, it does not depend on reliable message transfer and can therefore be used with unreliable datagram transport, e.g., *UDP*. It is designed to be as close to *TLS* as possible, so it is defined as a series of deltas to *TLS*. The main changes to the protocol due to lossy message transfer (when using *UDP*) are: a) Handling packet loss, b) reordering of messages, and c) message sizes. The concrete handling of these challenges are described in Chapter 5.

DTLS is a protocol composed of two layers: The lower layer is called the *DTLS Record Protocol*, which provides connection security that has two basic properties: a) The connection is private by using symmetric encryption. b) The connection is reliable by including a message integrity check. These options may be used alone, together or not at all. The protocol encapsulates four higher-level protocols [10]:

DTLS Handshake Protocol
: This protocol is used to negotiate the security parameters of a session later used for protected communication.

DTLS Change Cipher Spec Protocol
: The change cipher spec protocol exists to signal transitions in ciphering strategies. It is used to notify the receiving party that subsequent records will be protected using the negotiated security parameters.

DTLS Alert Protocol
: This protocol can be used at any time during the handshake and up to the closure of a session, signalling either fatal errors or warnings.

DTLS Application Data Protocol
: Application data messages are carried by the record layer and are fragmented, compressed, and encrypted based on the current connection state.

While *DTLS* has been defined in the *CoAP* draft to be mandatory [20], it is not fully optimal for resource-constrained networks as it (or more accurate *TLS*) was designed for traditional computer networks. For example, large buffers are needed to handle message loss by retransmitting the last flight of the handshake protocol or to store all the fragments when fragmentation must be applied due to smaller *Maximum Transmission Unit (MTU)*. Another problem is the use of X.509 certificates to achieve mutual authentication between the peers: such a certificate chain can easily grow to the size of a few kilobytes which leads to high fragmentation in networks with a small *MTU* resulting in a higher chance of message lost and need to retransmit the whole flight. To address this issue, the concept of *raw public key* certificates has been drafted in [44] and declared mandatory in the *DTLS* specification: By sending only the peer's public key and validate that public key using an out-of-band technique, a large part of the X.509 certificate chain could be cut and need

not to be sent over the resource-constrained network. A detailed description about this concept can be found in Section 5.4.

There are still great challenges to overcome, if *DTLS* should become a valuable technique for securing a constrained environment in the future: The protocol needs to be simplified, stripped off all unnecessary features to achieve a useful trade-off between security and a light-weight implementation suitable for a constrained environment.

3.4.3 Link-Layer Security

As we have seen security protocols applied at the *network* and *transport* layer, let us look at security at the *link* layer. The current *IoT* uses for communication primarily 6LoWPAN [26] which builds on the IEEE 802.15.4 link-layer. 802.15.4 link-layer security is the current state-of-the-art security solution for the *IP*-connected *IoT* [31]. It provides data encryption and integrity verification, which is achieved by a single pre-shared key used for symmetric cryptography applied to all outgoing packets while message integrity is realised by including a *Message Authentication Code (MAC)* in the packets. Its advantages are network protocol independence and hardware support for the cryptographic functions by 802.15.4 radio chips. Unfortunately, there is one big drawback with this security approach: The link-layer security only provides *hop-by-hop* security which implies that each node in the communication path must be trusted and host authentication is not possible. Additionally, messages leaving the 802.15.4 network will not be protected anymore by link-layer security mechanisms. Since end-to-end security cannot be achieved by using solely this approach, additional mechanisms are required as proposed in [31].

3.4.4 Physical Security

In the threat analysis, we identified several possible physical attacks to which things are exposed. One of the possible attacks is called *node capture*, where an adversary tries to gain full control over some node through direct physical access. It is usually assumed that this attack is "easy," as the nodes operate unattended and cannot be made tamper proof because they should be as cheap as possible [5]. But, Becher, Benenson, and Dornseif found out that serious attacks require an absence of captured node for a substantial amount of time, and therefore, countermeasures are possible against such node capture attacks [5]:

- Standard precautions for protecting microcontrollers from unauthorised access must be taken, e.g., by protecting the bootstrap loader password.
- Provide a hardware platform appropriate for the desired security level, which is up-to-date with the newest developments in embedded systems security. Sometimes, building additional protection around a partially vulnerable microcontroller might be reasonable.

- By monitoring nodes for periods of long inactivity, or noticing the removal of a node from the deployment area, the network can revoke the authorisation tokens of the suspicious node. Or, the node itself might erase all confidential data stored on it when suspecting a physical attack.

3.5 Roadmap

Although *DTLS* has not been designed with constrained nodes or networks in mind, it is thought to be usable in such environments. But, there are several challenges when it comes to implement *DTLS*. Hartke & Bergmann [15] present ideas for a constrained version of *DTLS* that is friendly to constrained nodes and networks.

As this part requires an advanced knowledge about the *DTLS* protocol, we advise the reader to study Appendix A prior to the following explanations.

3.5.1 Handshake Message Fragmentation

DTLS proposes the fragmentation of handshake messages to avoid *IP* fragmentation. This may help to send large *DTLS* records (such as key exchange and certificates messages) over 6LoWPAN where the physical layer *MTU* can be as small as 127 bytes, but at the same time this introduces a large overhead on the number of datagrams and bytes transferred: When no fragmentation must be applied to the handshake messages, the header data takes up only 34% compared to 55% when the *UDP* data size is limited to 50 bytes [15]. In addition, constrained networks are still prone to packet loss, which causes the implementation to provide buffers large enough to hold all received messages and losing one fragment forces the retransmission of the whole flight. This is a big problem for memory-constrained nodes and low-power networks.

To prevent message fragmentation or at least reduce the number of fragments that need to be transferred, the goal needs to be to minimize the total number of bytes that need to be sent. Possible solutions to this problem could be: a) Use an out-of-band mechanism to exchange large, static information, such as the certificate. Omitting the exchange of full X.509 certificates or only sending the *raw public key* would cut down the number of transferred bytes considerably. b) Use 6LoWPAN General Header Compression to compress *DTLS* messages. c) Use some *DTLS*-specific Stateless Header compression (described in more detail later).

3.5.2 Timer Values

DTLS leaves the choice of timer values (the time that needs to expire before a flight is retransmitted) to the implementation, but it suggests an initial timer value of 1 second and doubling the value each time a retransmit occurs. As Hartke & Bergmann show, certain key exchange or signature algorithms can require more time on constrained devices than the initial timer value of 1 second, leading to spurious retransmission. To avoid this from happening, it is important that, for constrained nodes and low-power, lossy networks, the initial timer value is adjusted in such a way that retransmissions only occur when a message is really lost. Unfortunately, there exist no guidance value for constrained environments; each individual network needs to be evaluated separately to find an appropriate value.

3.5.3 Connection Initiation

One big issue for nodes with very constrained memory is the complexity of the *DTLS* handshake protocol: Up to 15 handshake messages distributed over 6 flights are needed to establish a secure connection. This requires the node to keep the current state of the protocol and provide buffers for storing the received fragments and possibly retransmitting messages. Also, *DTLS* introduces, compared to *TLS*, two additional messages to the protocol that result in two additional flights during the handshake, which adds extra complexity to the protocol: The *HelloVerifyRequest* and the second *ClientHello* message containing the cookie to prevent *DoS* attacks. Hartke & Bergmann envision that the acceptance of *DTLS* as security protocol for embedded devices would significantly increase if a less complex connection initiation procedure with a smaller number of handshake messages was defined.

The handshake could be shortened to four flights with 11 handshake messages exchanged without losing the *DoS* roundtrip if the ciphersuite permits that the server remains stateless after the sending the *ServerHello* (i.e., `TLS_PSK_WITH_AES_128_CCM_8` would allow this, while `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` would not) and if the flight fits in one datagram. Figure 3.1 illustrates this idea: To allow the server to remain stateless, the client must provide the server with its own *ServerHello* message in the third flight, such that the server can establish the negotiated state and finish the handshake afterwards. By choosing a *Pre-Shared Key (PSK)* key exchange algorithm, the server's *ServerKeyExchange* message and all messages responsible for mutual authentication can be omitted. One could argue that the *HelloVerifyRequest* could be dropped entirely in this scenario, since the server does not send any large messages in its first flight and therefore cannot be abused by an attacker for amplification. Also, the server does not retransmit this first flight because it must remain stateless until receiving the client's second flight.

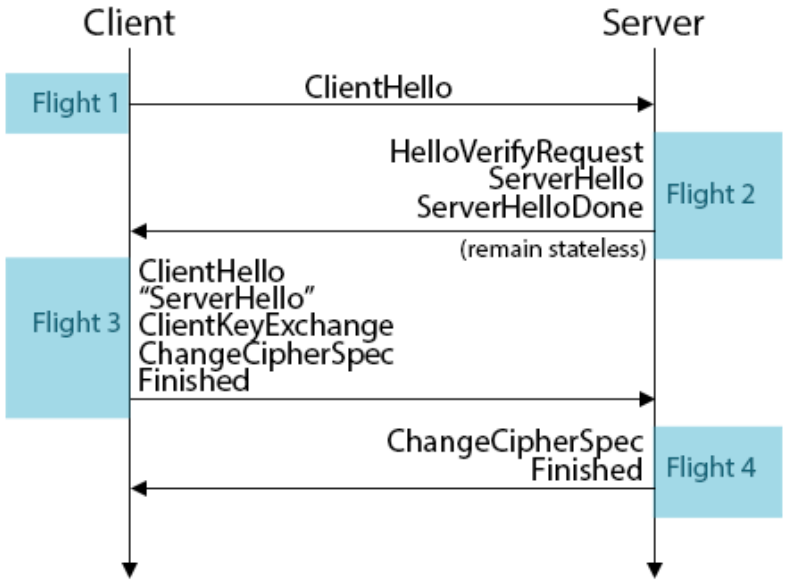


Figure 3.1: A DTLS handshake that needs only four flights (compared to the six for a normal handshake) [15].

3.5.4 Connection Closure

After a handshake has finished, the entities must still store around 80 bytes for the negotiated keys (with `AES-128-CCM`), sequence numbers and anti-replay window. If the connection needs to be resumable, the 48-byte master secret and the negotiated security parameters need to be stored as well. While this is considerably less memory needed as compared during a handshake, it still limits the amount of connections a constrained device can maintain at a given time. It is recommended that both parties keep the connection alive as long as possible, such that the expensive handshake needs to be executed as little as possible [15]. When one party runs out of resources, it can close the connection by sending a closure *Alert*. As long as no closure *Alert* is received, both parties assume that the connection is still alive until its application data is rejected, in which case a new connection must be established.

3.5.5 Application Data Fragmentation

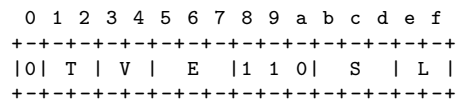
DTLS only supports fragmentation for handshake messages, but not for application data. If *IP* fragmentation wants to be avoided, the application layer protocol needs to handle this. *CoAP* supports block-wise transfer for this case. Wrapping a *CoAP* message into a *DTLS* record layer to provide security generates an overhead of 13 bytes for the record header. Additionally, *Authenticated Encryption with Associated Data (AEAD)* ciphers such as `AES-128-CCM` adds another 16 bytes for the explicit nonce and the authentication tag. Thus, leading to an overall overhead of 29 bytes for each encrypted *DTLS* packet. With packet sizes of 80 bytes, this takes a considerable portion of the available packet size away (64% left for application data) [15]. There are two possible optimisations to this issue: a) Omit the *DTLS* version field where it is implicitly clear. Since the version is negotiated in the handshake, there is no need to specify it in every record afterwards. This would save us 2 bytes. b) Elide the `nonce_explicit` field when `AES-CCM` is used. This 8-byte field is set to the 16-bit `epoch` concatenated with the 48-bit `seq_num` [25]. This means that the epoch and sequence number are unnecessarily included twice in the record header. This would save another 8 bytes and together with stateless header compression (explained next) the number of bytes left for the application data can be incremented significantly.

3.5.6 Compression

Stateless Header Compression

Hartke & Bergmann propose stateless header compression which compresses the header of records and handshake messages. It is lossless and is done without building any compression context state. This compression builds upon the premise that certain fields in the header only need a fraction of the space that is allocated in the fixed version: E.g., the `epoch` field uses 2 bytes, but during a normal handshake only the values 0 and 1 are needed which could be stored using only 1 bit. The same is true for the `seq_num` field which uses 6 bytes while normally could be stored using only a fraction thereof.

Records are compressed by specifying the header fields using a variable number of bytes. A prefix is added in front of the structure to indicate the length of each field or specify the value of the field directly. If the value is specified directly, the field itself can be omitted. The format of the prefix is as following:



Note that the set bits at position 0, 8, 9 and a are used to safely distinguish the prefix from the uncompressed header. The fields in the prefix are defined as follows:

T: Type Field

0 - Content Type 20 (ChangeCipherSpec)
1 - 8-bit type field
2 - Content Type 22 (Handshake)
3 - Content Type 23 (Application Data)

V: Version Field

0 - Version 254.255 (DTLS 1.0)
1 - 16-bit version field
2 - Version 254.253 (DTLS 1.2)
3 - Reserved for future use

E: Epoch Field

0 - Epoch 0
1 - Epoch 1
2 - Epoch 2
3 - Epoch 3
4 - Epoch 4
5 - 8-bit epoch field
6 - 16-bit epoch field
7 - Implicit – same as previous record in the datagram

S: Sequence Number Field

0 - Sequence number 0
1 - 8-bit sequence_number field
2 - 16-bit sequence_number field
3 - 24-bit sequence_number field
4 - 32-bit sequence_number field
5 - 40-bit sequence_number field
6 - 48-bit sequence_number field
7 - Implicit – number of previous record in the datagram + 1

L: Length Field

0 - Length 0
1 - 8-bit length field
2 - 16-bit length field
3 - Implicit – last record in the datagram

As an example, let us look at the at the first handshake message which will be sent by the client and how much can be saved by using stateless header compression: The uncompressed header will contain 13 bytes: 1 for the content type (`0x16`), 2 for the version (`0xfeff`), 2 for the epoch (0), 6 for the sequence number (0) and 2 for the length (`0x0038`).

The compressed header on the other hand would only need 3 bytes: 2 for the prefix and 1 byte for the length field (56 can be represented using 1 byte only). All other fields can be expressed in the prefix directly and are therefore omitted in the compressed header.

```
Uncompressed Header:
+--+--+--+--+--+--+--+--+--+--+--+--+--+
|16|fe ff|00 00|00 00 00 00 00 00|00 38|
+--+--+--+--+--+--+--+--+--+--+--+--+--+

Prefix (2 bytes):
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| T | V |  E  |1 1 0|  S  | L |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|1 0|1 0|0 0 0|1 1 0|0 0 0|0 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Compressed Header:
+--+--+--+
|50 C1|38|
+--+--+--+
```

So, in this example we would have saved 10 bytes for the record header alone. For the handshake message header there exists similar compression rules which can be found in [15].

6LoWPAN Compressed DTLS

Devices in the *IoT* can use the 6LoWPAN to compress the long *IP* layer headers. Raza, Trabalza and Voigt propose a technique similar to the presented stateless header compression by noting that the 6LoWPAN header compression can also be used to compress the security headers of *DTLS* [32]. Unlike Hartke & Bergmann, this compression does not only try to compress the record and handshake headers but the handshake messages as well. With this approach a significant number of bits can be saved: The record header, which is included in all messages, can be compressed by 64 bits (62% of the record header) for each message, or the size of the *ClientHello* can be reduced from 336 bits to 264 bits (23% compression) [32].

3.5.7 RESTful DTLS Handshake

CoAP, with its support for reliable message transmission and blockwise-transfer [7], could be used to transfer *DTLS* handshake messages, instead of the complex *DTLS* handshake protocol. With this, the need for *DTLS* to support retransmission, fragmentation and handling of reordering could be omitted and just the handshake logic needed to be implemented [15]. *CoAP* is designed for applications following the REST architectural style. So, the *DTLS* connection is modelled as a *CoAP* resource which is created when a client wants to initiate a connection and updated to modify the state and parameters

of the connection. Figure 3.2 illustrates this idea: The client `POST`s to the well-known URI requesting the server to create a new session resource. The server responds with the Verify response code (not yet defined) and *HelloVerifyRequest* message in the payload to which the client responds with the same `POST` containing the cookie. After the server having created the resource, the client requests the server to change session parameters by applying the `PATCH` method to the resource. The `DELETE` method can be used to close a current connection and free all resources related to the session.

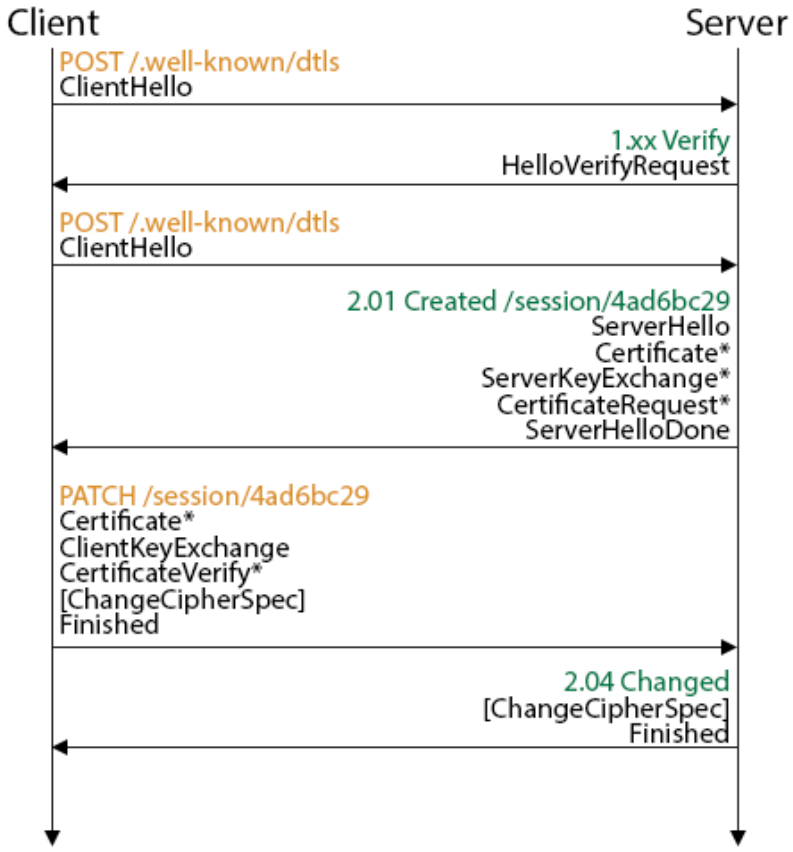


Figure 3.2: The message flights of a RESTful DTLS handshake [15]. The message denoted by an * are optional messages.

3.5.8 Templates

Hartke & Bergmann also proposed templates which could help make building and parsing the bodies of *Certificate*, *ServerKeyExchange* and *ClientKeyExchange* simpler when *Elliptic Curve Cryptography (ECC)* is used. Instead of decoding and encoding the elliptic curve points according to the ASN.1 standard [11], which needs lots of memory consuming code, the static body parts of the message can be used to achieve this more easily. Unfortunately, after several tests, we had to detect that this cannot be achieved that easily. Although the elliptic curve points have a fixed length, in the handshake message they must be encoded as a *Distinguished Encoding Rules (DER)* integer value. By definition, if the leftmost bit of an integer value is 1, an extra byte needs to be added to the encoded value resulting in different lengths of the encoded points. So, using total static pre-composed messages cannot work, but a more variable approach needs to be found, if this templates should be used in constrained environments.

We have presented several ideas on how to make the *DTLS* protocol more suitable for constrained environments. Apart from the stateless header compression, changes to the *DTLS* protocol specification need to be performed with greatest care. So far, no in-depth security analysis has been executed regarding these possible changes, which will be essential before even thinking about implementing these ideas.

As we have now seen the different aspects of a secured *Internet of Things (IoT)* in combination with *Constrained Application Protocol (CoAP)*, we present the first step towards such a secured environment: We implemented *Datagram Transport Layer Security (DTLS)* to secure *CoAP* during the operational phase.

4 DTLS for Californium

This chapter describes the design decisions which have been made to integrate *DTLS* into *Californium (Cf)*. *DTLS* is a complex security protocol and requires detailed knowledge in various areas. We try to explain the most important features when they are needed, but the reader is advised to read Appendix A previously to the following chapters, to gain an overview about the *DTLS* handshake protocol and the used concepts.

4.1 Overview

As mentioned in Chapter 2, *Cf* uses a layered architecture to add functionality to the communication stack. Therefore, adding security to this stack implies just adding another layer that takes care of this. The secured communication stack is shown in Figure 4.2. Compared to the unsecured communication stack in Figure 4.1, the *UDPLayer* has been replaced by the *DTLSLayer*. This decision implies that the *DTLSLayer* is responsible for adding the security feature as well as executing the job of the *UDPLayer* of sending the message over the network. This decision of replacing the lowest layer instead of inserting the *DTLSLayer* above the *UDPLayer* was mainly motivated by the newly introduces message formats of *DTLS*: The current layer interface can only handle *CoAP* messages and to make message passing between the layers possible for both *CoAP* and *DTLS* messages, would have needed a lot of changes to the layer architecture. So, as long as the *DTLSLayer* is the lowest layer in the communication stack, the other layers must not be able to handle *DTLS* messages as the *DTLSLayer* will only exchange *CoAP* messages between the upper layers. Another reason for this decision is *CoAP*'s binding of the `coaps` URI scheme and the default security port to *DTLS*: There exist different default ports for the `coap` and `coaps` URI scheme and therefore, we need two different communication stacks—or at least two different lowest layers—listening on different ports. Thus, the expected message format on the different ports is known and the lowest layer must not be able to distinguish the formats to be able to parse the datagram correctly. So, currently, the *UDPLayer* is responsible for parsing *CoAP* messages listening on port 5683 and the *DTLSLayer* for *DTLS* messages on the standard security port, which has not yet been standardised [37].

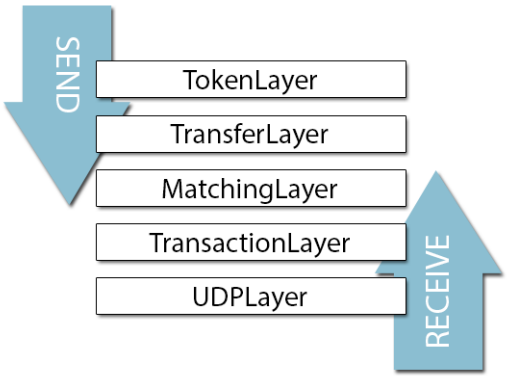


Figure 4.1: The individual layers of an unsecured CoAP communication stack.

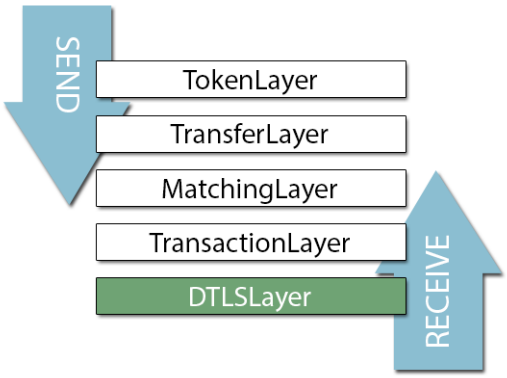


Figure 4.2: The individual layers of a secured CoAP communication stack.

4.2 DTLS Layer

Figure 4.3 provides a schematic overview of the whole *DTLS* architecture. The different concepts used in this graphic are explained in the following chapters.

The DTLS Layer works roughly as follows:

1. When the *DTLSLayer* receives a *CoAP* message from the upper layer, it checks whether a *DTLSSession* exists with the peer:
   a) If an active *DTLSSession* is found, the *DTLSLayer* sends the *CoAP* message as encrypted application data.
   b) If a *DTLSSession* exists with the peer that has been closed already, the *DTLSLayer* initialises a *Handshaker* to execute an abbreviated handshake to resume the session.
   c) If there is no such *DTLSSession*, a full handshake must be executed.
   d) The newly initialised *Handshaker*, which is either of type `ClientHandshaker` or `ResumingClientHandshaker` (as listed in Figure 4.5), returns the *DTLSFlight* to kick-start the handshake protocol which is sent to the peer by the *DTLSLayer*.
   e) After a successful handshake, the *CoAP* message will be sent as application data protected with the newly negotiated security parameters.
2. When a *UDP* datagram is received, the packet is parsed into one or more *Records* and each is processed in the following way:

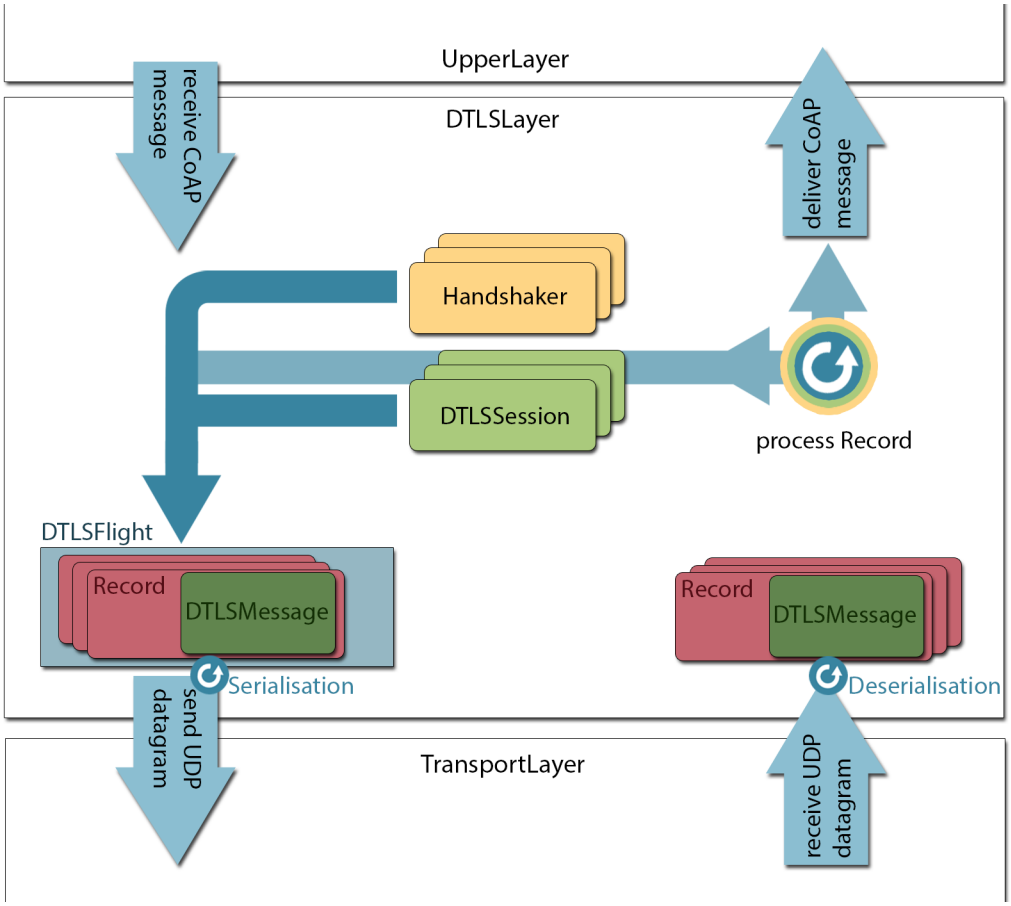


Figure 4.3: Schematic overview of the DTLS layer.

a) If the *Record* contains application data, it is decrypted with the current read state defined in the *DTLSSession* associated with this peer and the resulting *CoAP* message is delivered to the upper layer. If no such *DTLSSession* exists, the message is discarded.

b) If the content of the *Record* is of type handshake, the *DTLSLayer* forwards the message to the appropriate *Handshaker* associated with the peer. The *Handshaker* processes the message according to the current handshake protocol state and potentially returns a *DTLSFlight*, which is sent by the *DTLSLayer* to the peer.

To summarise, the *DTLSLayer* is responsible for the following tasks:

- Receiving and delivering *CoAP* messages from the upper layer.
- Sending and receiving of *UDP* datagrams and handling the serialisation.
- Associating endpoint addresses with *DTLSSessions*, *Handshakers* and *DTLSFlights*.
- Handling the retransmission timers and retransmitting the corresponding flight when the timer expires.

4.3 Handshaker

The *Handshaker* is responsible for the handshake logic and keeps track of the state of the current handshake execution between two endpoints. So, for each new handshake between two peers, a new `Handshaker` object must be generated. The *Handshaker* is driven asynchronously by the handshake messages delivered to it by the *DTLSLayer*. The specific logic is implemented in the subclasses of the abstract class `Handshaker` by providing implementations for the two abstract methods:

`DTLSFlight processMessage(Record message)`
: This method is called by the *DTLSLayer* on the corresponding `Handshaker` object when a handshake message is received that belongs to the handshake execution associated to this handshaker. The handshaker processes the received message according to its handshake message type and reacts corresponding to the current state of the handshake protocol. It returns the next *DTLSFlight* of the handshake protocol, if the currently processed message marks the last of the peer's current flight. If an error occurred during the processing of the message, a `HandshakeException` is thrown which will be caught and handled by the *DTLSLayer*.

`DTLSFlight getStartHandshakeMessage()`
: This method returns a `DTLSFlight` containing the handshake messages needed to kick-start or resume a handshake, which will then be sent by the *DTLSLayer*.

Figure 4.5 depicts the four types of handshakers that implement these two methods:

`ClientHandshaker`
: The `ClientHandshaker` executes a full handshake from the point of view of the client. It will only allow to receive the following handshake messages: a) *HelloRequest*, b) *HelloVerifyRequest*, c) *ServerHello*, d) *Certificate*, e) *ServerKeyExchange*, f) *CertificateRequest*, g) *ServerHelloDone* and h) *Finished*. All other received messages will result in an aborted handshake. Its kick-start flight will contain a *ClientHello* message with an empty session identifier.

`ResumingClientHandshaker`
: The `ResumingClientHandshaker` executes an abbreviated handshake from the point of view of a client. It inherits certain client functionality by extending the `ClientHandshaker` class but overrides the two abstract methods. It will only process these two messages: *ServerHello* and *Finished*. Its first flight will contain a *ClientHello* message having a session identifier of a previous finished handshake.

`ServerHandshaker`
: The server's counterpart to the client's full handshaker. It will only process these handshake messages: a) *ClientHello*, b) *Certificate*, c) *ClientKeyExchange*, d) *CertificateVerify* and e) *Finished*. To ask the client to start a new handshake, the `ServerHandshaker` sends a *HelloRequest* message.

`ResumingServerHandshaker`
: The server's counterpart to the client's abbreviated handshaker. It overrides the `processMessage()` method of the `ServerHandshaker` and can only process *ClientHello* and *Finished* messages.

The abstract class `Handshaker` provides the general functionality and fields which will be used by all types of handshakers when executing a handshake:

- Generation of the master secret from a given premaster secret and calculating the corresponding read and write keys.
- Handling reordering of messages by deciding whether the received message is the next one that can be processed.
- Wrapping handshake messages into record layers, handling the sequence number and epoch, and apply fragmentation if needed.
- Reassembling fragmented messages.
- Loading of the private key and the according certificate which will be used in the handshake protocol when authentication is required.
- Providing buffers to queue messages and store fragmented messages.
- Holding the shared keys when the *PSK* key exchange mode is used.

4.4 DTLS Session

The class `DTLSSession` represents a *DTLS* session between two peers. It keeps track of the current and pending read/write states, the current epoch and sequence number for the record that will be sent next, and the key exchange mode. Additionally, it holds the negotiated *Master Secret* which can be used in abbreviated handshakes to derive fresh keys. The *DTLSLayer* stores all sessions that have been established during a successful handshake.

4.5 Record

The *DTLS* protocol exchanges *Records*, which are used to encapsulate the data that needs to be exchanged (e.g., handshake messages or application data). Depending on the current connection state, the *Record* is compressed, padded, *MAC*ed and encrypted. Each *Record* has a content type field that specifies the encapsulated protocol (i.e., *handshake*, *alert*, *change cipher spec*, or *application data* protocol), which is treated by the `Record` class as opaque data that needs to be dealt with by the specified higher-level protocol [10].

4.6 DTLS Message

A *DTLS* message represents a generalisation of the four protocol message types which are encapsulated in the *Record* layer (see Section 3.4.2). The corresponding interface `DTLSMessage` is used by the *Record* layer to hold the opaque protocol data. It is implemented by the four content types (see Figure 4.4):

`Change Cipher Spec`
: The *Change Cipher Spec* protocol is used to signal transitions in ciphering states. The message is used by both client and server to notify the peer that all following records will be protected under the negotiated security parameters. When receiving such a message, the read *current* state is changed to the read *pending* state.

`Alert`
: *Alert* messages are used to notify the peer about problems or errors in the current handshake execution. *Fatal* alerts result in an immediate termination of the current connection while *Warning* alerts need to be evaluated: The recipient must judge whether it is still safe to continue the current handshake given the specific warning.

`Application Data`
: *Application Data* messages are treated by the *Record Layer* as transparent data. It is compressed and encrypted based on the current encryption state. The *DTLS Layer* parses the decrypted application data and returns the *CoAP* message to the upper layer.

`Handshake`
: The *Handshake Protocol* is responsible for negotiating a session and generating shared secrets which can then be used by the *Record Layer* to protect the application data. To complete a full handshake (see Appendix A for the detailed protocol), several different handshake message types are needed; see Figure 4.4 for all the message types.

The `DTLSMessage` defines one method which must be implemented by all four subtypes:

`public byte[] toByteArray()`
: This method returns the raw binary representation of the message which is then used by the *Record* layer protocol for potential compression and encryption.

Please note that each implementation of the `DTLSMessage` interface must also provide a `fromByteArray()` method, but as this method is `static`, it is not defined in the interface.

4.7 DTLS Flight

A *DTLS Flight* represents a handshake protocol flight containing all handshake messages that need to be sent to the peer in the coming flight. See the Figure A.1 in Appendix A for an example of a flight. One flight contains of at least one message and needs to be retransmitted until the peer's next flight has arrived in its total. Although each flight of messages may consist of a number of messages, they should be viewed as monolithic for the purpose of timeout and retransmission.

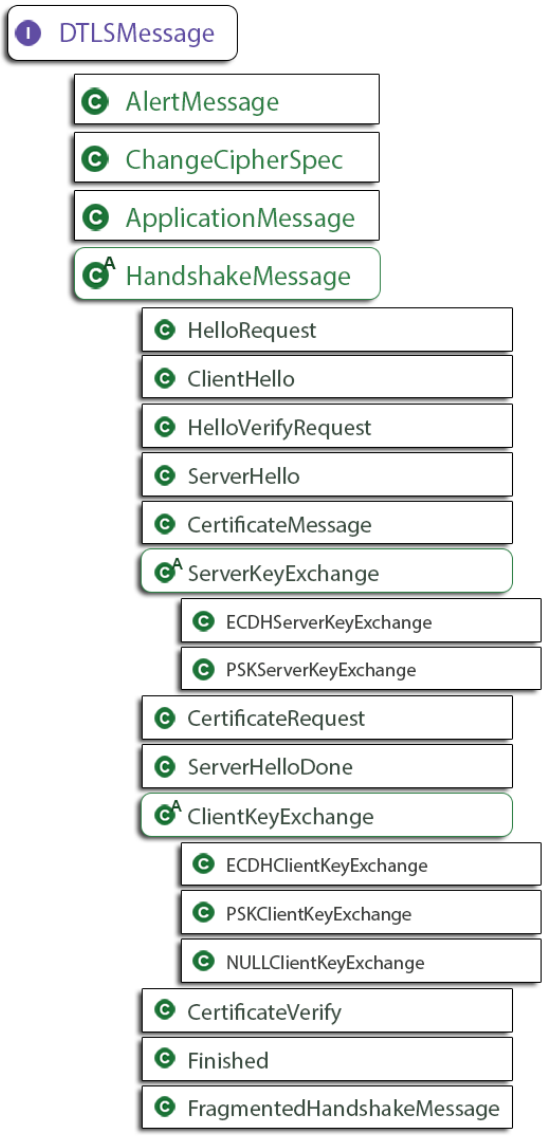


Figure 4.4: The interface `DTLSMessage` and its implementations. Classes denoted by an "A" are abstract classes.

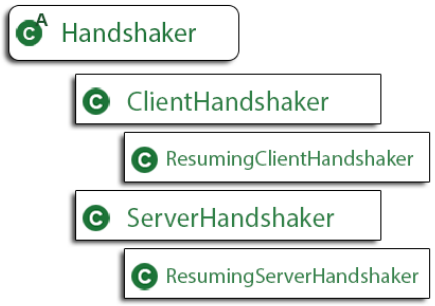


Figure 4.5: The abstract class `Handshaker` and its subclasses.

5 Implementation

When we were planning the design and implementation phase, we were faced with one fundamental decision: Should we implement *DTLS* from scratch or should we take an existing implementation of *TLS* and adapt it to the additional needs of *DTLS*. Such an adaptation has been implemented by N. Modadugu and E. Rescorla. It required adding about 7000 lines of additional code to the *OpenSSL* base distribution [27], which implements *TLS*. We decided not to follow this road and instead implement *DTLS* from scratch. This decision was mainly influenced by two factors: a) Changing an existing *TLS* framework such that the result is a light-weight *DTLS* implementation which can be used in a constrained environment seems rather unlikely to be successful due to the initial overhead. b) Although this *DTLS* implementation for *Cf* focuses on unconstrained environments and may never be used in constrained environments, as porting Java code to C code is quite hard, we still hope to gain something out of this: Namely *experience* about the design, the implementation and potential optimisations of the protocol. For readers interested in the source code, it is publicly available on Github[1].

5.1 DTLSLayer

As we have already mentioned in Chapter 4, the `DTLSLayer` is the lowest layer in the communication stack because it is responsible for sending and retrieving datagrams from the transport layer that resides beneath the application layer in which *Cf* resides. Therefore, it extends *Cf*'s abstract class `Layer` and not like the upper layers the class `UpperLayer`. An `UpperLayer` has the additional method `sendMessageOverLowerLayer()`. Although it would be possible to add a layer below the `DTLSLayer` in the communication stack, we decided that it must be the lowest layer whenever it is added to the stack. This decision has mainly been made because *DTLS* introduces new message formats. As long as this can be handled at the lowest layer, it has no influence on the other layers. But if the `DTLSLayer` should have been interchangeable with upper layers, the interface for passing message between the layers would have needed a lot of changes.

One of the main functionalities of the `DTLSLayer` is to distribute the received messages to the corresponding `Handshaker` object and to store the `DTLSSession` objects. For this task, it has two `Maps` that store `Handshakers` and `DTLSSessions` according to the peer's

[1] https://github.com/mkovatsc/Californium/tree/security

address. Whenever a message is received, the layer checks whether there exists an entry according to the peer's address. If neither a `DTLSSession` nor a `Handshaker` exist, only two handshake messages are allowed to be received:

ClientHello
: The peer wants to start a fresh handshake. Therefore, this entity will act as the server: The `DTLSLayer` initialises a `ServerHandshaker` and `DTLSSession`, stores them and forwards the message to the `ServerHandshaker` for processing.

HelloRequest
: The peer indicates that it wants this entity to start or resume a handshake. Therefore, this entity will act as the client: The `DTLSLayer` initialises a `ClientHandshaker` and `DTLSSession`, stores them and forwards the message to the `ClientHandshaker` for processing.

All other messages will be discarded. If a `DTLSSession` exists, but no `Handshaker`, the handshake must have been finished successfully and the session holds all the information needed to read the sent application data: The message is decrypted, the protected *CoAP* message extracted and forwarded to the upper layer. If an associated `Handshaker` exists, the layer forwards the received message to it for processing. Whenever a *CoAP* message is passed down from an upper layer, the `DTLSLayer` must check whether a session exists with the endpoint. If so, the message is sent as protected application data, otherwise a new handshake is started: A `ClientHandshaker` is initialised and starts the handshake by invoking the `getStartHandshakeMessage()`. The *CoAP* message is stored in the `ClientHandshaker` and sent as application data once the handshake is finished. Of course, there are more cases that must be distinguished, but those described are the most common ones and should suffice to illustrate the responsibilities of the `DTLSLayer`.

As can be seen by these explanations, the `DTLSLayer` is symmetric. This means, that we do not need different stacks for client and server endpoints but both instances can be handled inside the same layer. This symmetry is essential because in machine-to-machine applications endpoints may act as both client and server at the same time.

5.2 Cryptography

During the *DTLS* handshake and securing application data, a lot of different cryptographic algorithms are needed, such as for: a) Digital signatures, b) encryption, c) certificates and certificate validation, d) message digests (hashes), e) secure random number generation, and f) key generation and management.

Luckily, the *Java Cryptography Architecture (JCA)* provides a large set of APIs which help integrating security into application code. Whenever possible, we relied our implementation on this highly tested and maintained security platform, rather than trying to implement such difficult cryptographic algorithms by ourselves. However, some features of *DTLS* depend on emerging standards that are not yet implemented.

5.2.1 Native Java Support

JCA[2] defines so called *Engine Classes* which provide the interface to a specific type of a cryptographic service. We illustrate the use of such an *Engine Class* with the `Signature` class: To use a specific digital signature algorithm of *JCA*, the application simply requests a particular type of object (here `Signature`) and a particular algorithm (such as `SHA1withECDSA`) and gets an implementation from one of the installed security providers. Concretely, this is done by using one of the `Signature getInstance()` static factory methods (line 1 in Listing 5.1). This static method is part of each *Engine Class* and is the starting point for each provided algorithm.

Before, we mentioned the concept of a *Security Provider* which needs explanation as well: A *Security Provider* implements some or all parts of Java Security. Each *Java Development Kit (JDK)* installation has one or more providers installed and configured by default. So, if no specific provider is selected, the installed providers are searched through for a requested service (in our case `SHA1withECDSA` for `Signature`) until one is found that supplies a concrete implementation thereof.

The rest of the example code in Listing 5.1 is a straightforward example on how to produce a signature of some given data. The beauty in this approach lies within the fact that if we need another digital signature algorithm, only the first line needs to be changed while the rest of the code remains untouched. Please note that the selections of the algorithm instances are not hardcoded in the implementation but depend upon the negotiated ciphersuite.

Listing 5.1: Example Java code for the use of JCA.

```
1 // get the required signature algorithm
2 Signature signature = Signature.getInstance("SHA1withECDSA");
3
4 signature.initSign(privateKey);
5 signature.update(new byte[] {});
6
7 // the signature of an empty byte array
8 byte[] signatureEncoded = signature.sign();
```

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html

5.2.2 Own Implementation

As mentioned earlier, certain required cryptographic algorithms for *DTLS* are not yet implemented in *JCA*. Therefore we had to decide whether to use an external library (e.g., *Bouncy Castle*[3]) or to implement the missing functionality by ourselves. For the sake of future use of this implementation in constrained environments, we decided to implement it ourselves which should result in a more lightweight solution than including an external library. The following algorithms had to be implemented:

Hash-Based Message Authentication Code (HMAC)

HMAC is a mechanism for message authentication using cryptographic hash functions, such as *MD5* or *SHA-256*, in combination with a secret shared key [22]. Our implementation makes use of the available implementations for the underlying hash functions and needs only to append the resulting byte streams correctly according to the protocol. The test vectors supplied in [22] have all been passed.

Pseudorandom Function (PRF)

The *PRF* is defined in [10] and plays a crucial role in generating the *Master Secret*, key material, or `verify_data` needed in the *Finished* message used in the *DTLS* handshake. The *PRF* is defined as follows:

```
PRF(secret, label, seed) = P_<hash>(secret, label + seed), with
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...

where + indicates concatenation.

A() is defined as:
  A(0) = seed
  A(i) = HMAC_hash(secret, A(i-1))
```

`P_<hash>` denotes a data expansion function that uses a single hash function to expand a secret and seed into an arbitrary quantity output. *HMAC* uses this said hash algorithm as described above. The *PRF* can be used in the following manner:

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
                    [0..47];
```

Counter with CBC-MAC (CCM)

CCM (Counter with Cipher Block Chaining Message Authentication Code) is an authentication encryption block cipher mode designed to provide both authentication and confidentiality [42]. It is only defined for block ciphers with block length of 128 bits, such as *Advanced Encryption Standard (AES)*. The ciphersuites named **_WITH_AES_128_CCM_8* uses *AES-128* as the block cipher and uses 8 octets (64

[3] http://www.bouncycastle.org/

bits) for authentication resulting in a ciphertext that is 8 octets longer than the corresponding plaintext. For the underlying cipher the available implementation of *JCA* is used (e.g., *AES*). The test vectors defined in [42] have all been passed.

It is important to note that certain native Java cryptographic functions, upon which this implementation relies, are not available before Java 7. Mainly, this includes all functions using *Elliptic Curve Cryptography (ECC)*.

5.3 Ciphersuites

A ciphersuite is a named combination of key exchange, authentication, encryption, and message authentication algorithms. It is used to negotiate the security settings under which the *DTLS* handshake will be executed. The *CoAP* draft declares two ciphersuites to be mandatory: `TLS_PSK_WITH_AES_128_CCM_8`, as specified in [12] and [25], and `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`, as specified in [6], [10] and [25].

5.3.1 TLS_PSK_WITH_AES_128_CCM_8

This ciphersuite uses the *PSK key exchange algorithm* to establish the *Master Secret*. This results in a simplified handshake protocol (compared to the one described in Appendix A): Since both parties have pre-shared keys exchanged in the past, the need for mutual authentication can be omitted (no *ServerCertificate*, *CertificateRequest*, *ClientCertificate* and *CertificateVerify* message needed) because trust in the other party must have been established beforehand. By *trust* we mean basically *authenticity*: If the peer possesses the pre-shared key, it must have been distributed previously in a secure way (e.g., during the *bootstrapping* phase) where the peer must have authenticated itself. An attacker may indeed impersonate the peer, but has no way of guessing the pre-shared key correctly and finishing the handshake successfully, rendering authentication unnecessary. Additionally, the *ServerKeyExchange* message can also be left out, unless there exist multiple pre-shared keys between the client and the server, and the server wants to help the client in selecting the appropriate one by providing a *PSK identity hint*. If no such hint is provided by the server, the client selects one key and indicates this by including a *PSK identity* in the *ClientKeyExchange* message. The storing and retrieving of pre-shared keys has been solved so far in a very simple way: A `HashMap` which stores keys according to *PSK identities*.

```
protected static Map<String, byte[]> sharedKeys =
        new HashMap<String, byte[]>();
```

Currently, the server never sends a *PSK identity hint* and the client's *PSK identity* is determined using the `Properties` file (see 5.8). This solution with hardcoded pre-shared keys in the source code might work for testing purposes but is obviously not

suitable for larger and dynamically changing networks. A more flexible solution must be found to add pre-shared keys. This problem needs to be solved during the *bootstrapping* phase of the network and since this implementation focused on the use of *DTLS* during the *operational* phase, finding solutions has been ignored so far and will be subject of future works.

While generating the *Master Secret* from a given *Premaster Secret* is the same for all ciphersuites, the generation of the *Premaster Secret* is specific to the used ciphersuite. The *Premaster Secret* is formed as follows: If the *PSK* is N octets long, concatenate a `uint16` with the value N, N zero octets, a second `uint16` with the value N, and the PSK itself [12]. To illustrate this, if the *PSK* is { `0x01, 0x02, 0x03` } the *Premaster Secret* will become:

```
{ 0x00, 0x03, 0x00, 0x00, 0x00, 0x00, 0x03, 0x01, 0x02, 0x03 }
  <--- N -->  <--- N zero --->  <--- N -->  <---- PSK ----->
```

The 2-byte identifier of the ciphersuite used in the *Hello* messages is `0xC0,0xA8` as specified by the *Internet Assigned Numbers Authority (IANA)* [40].

5.3.2 TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8

This ciphersuite uses the *Elliptic Curve Diffie-Hellman (ECDH)* key agreement to generate the *Premaster Secret*, and *Elliptic Curve Digital Signature Algorithm (ECDSA)* as the authentication mechanism. For detailed explanations about this handshake and the used *ECC* mechanisms, please refer to Appendix A and B. Luckily, Java provides native support for almost all mechanisms that are needed to use this ciphersuite:

- Generating ephemeral *ECDH* key pairs used for the key agreement.
- Executing the key agreement protocol and generating the *Premaster Secret*.
- Creating an *ECDSA* signature and validating it.

We only had to implement two things on our own:

Encoding

To run the *ECDH* key agreement algorithm with ephemeral keys, both the client and the server must supply the other party with their public key which represents a point on the negotiated elliptic curve. When the key exchange object is serialised, this point must be encoded according to the conversion routine of ANSI X9.62 [6] which is not supported by Java. Since the conversion is straightforward (and therefore poses low risk to a faulty implementation), we decided to implement it ourselves instead of relying on an external library.

Named Curve Parameters

A named curve specifies the domain parameters for one specific elliptic curve. This comes in handy when the server sends its ephemeral public key and the used elliptic curve parameters in the *ServerKeyExchange* message: Instead of sending the complex domain parameters in its entirety, the server needs only to transmit the named curve identifier. Although Java provides the `ECParameterSpec` class which is designed to hold the domain parameters of an elliptic curve, a lookup table for named curve identifiers does not exist. Therefore, we decided to create such a static lookup table which maps identifiers to parameters:

```
public final static Map<Integer, ECParameterSpec>
                NAMED_CURVE_PARAMETERS;
```

The elliptic curve domain parameters have been defined by *Standards for Efficient Cryptography Group* and can easily be used to initialise the needed `ECParameterSpecs` [38].

The 2-byte identifier of the ciphersuite has not yet been defined [40] by *IANA*, but for now we used the next available value `0xC0,0xAC`.

5.4 Certificates

To achieve authentication, the *DTLS* handshake protocol specifies the *Certificate* message. It contains a certificate chain of X.509 certificates where the first certificate in the chain contains the peer's public key. To make the loading as easy as possible we decided to use Java *keytool* which is a key and certificate management tool that manages a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates[4].

For our purpose, there exist two keystore files: The keystore containing a private key and a certificate chain with the first certificate having the corresponding public key, and the truststore that contains all trusted certificates. Appendix C shows how to use the `keytool` and `openssl` commands to generate a keystore containing a certificate chain signed by a trusted certificate authority. Loading of the certificate chain and private key can then be easily achieved by the use of *alias* given to the private key and certificate chain. The Java code in Listing 5.2 illustrates this.

As described in [44], the use of *raw public keys* could be a good solution for *DTLS* used in constrained environments: Instead of sending a large X.509 certificate chain to achieve authentication, an entity can send only its public key in the *Certificate* message while the validation of the public key is achieved by an out-of-band method. This case can be handled almost the same way as sending the whole certificate chain. The loading of

[4] http://docs.oracle.com/javase/7/docs/technotes/tools/windows/keytool.html

Listing 5.2: Example Java code for loading the private key and certificate chain.

```
1 KeyStore keyStore = KeyStore.getInstance("JKS");
2 // path loaded from Californium.properties file
3 InputStream in = new FileInputStream("path/to/keyStore");
4 keyStore.load(in, "password");
5
6 Certificate[] certificates = keyStore.getCertificateChain("alias");
7 PrivateKey privateKey = keyStore.getKey("alias", "password");
```

the certificate through the keystore can be done the same way as already explained and extracting the public key can be achieved easily:

```
byte[] rawPublicKey = certificateChain[0].getPublicKey().getEncoded();
```

The `getEncoded()` method applies the right encoding to the public key, namely using the ASN.1 *DER* according to the X.509 standard [9], and therefore the *raw public key* can be added to the payload of the *Certificate* message without any further modifications.

Please note that the proposal for the use of *raw public keys* still had the status of an Internet-Draft when this thesis was written and therefore the concept has not been fully defined. Up to draft number 4, it has not been defined whether the *raw public key* adopts the format structure of the *Certificate* message or whether the payload of the handshake message only consist of the *raw public key*. The authors of this Internet-Draft stated that this is not clear yet and needs to be specified in a later version of the draft.

5.5 Error Handling

Error handling is a crucial part of the *DTLS* handshake protocol, since there are several points where failures must be expected during the protocol. In general, there are two possible source of errors:

Handshake Exceptions

These type of exceptions are thrown intentionally during the handshake. Potential errors can occur when: a) Authentication of the cipher failed. b) The received certificate chain could not be validated. c) A signature could not be verified. d) No *Pre-Shared Key* was found. e) Wrong *ECDH* parameters used by peer. f) The `verify_data` in the *Finished* message does not match. g) Unknown `HelloExtension` type received. h) Unsupported version negotiated. i) No suitable ciphersuite found. j) Unexpected handshake message received.

All these errors lead to a `FATAL Alert` message and therefore to the abortion of the current handshake. In all cases, a `HandshakeException` must be thrown with the error-specific `AlertMessage`, e.g., when the cipher could not be authenticated (Listing 5.3).

Unexpected Exceptions

Exceptions suitable for this category happen unexpected and can occur at any particular point in the protocol. The *Java Virtual Machine (JVM)* throws the corresponding exception when such an error happens. The most common exception is the `NullPointerException` which mainly occurs due to parse errors.

Listing 5.3: Example Java code illustrating how to react upon failed cipher authentication.

```
1 String message = "The encrypted message could not be authenticated";
2 AlertMessage alert = new AlertMessage(AlertLevel.FATAL,
    AlertDescription.BAD_RECORD_MAC);
3 throw new HandshakeException(message, alert);
```

All possibly thrown exceptions are caught at a single point in the *DTLSLayer*. To make the *DTLSLayer* stable against unexpected errors, the whole method that receives the datagram, parses the message and reacts according to the message, is surround with a *try/catch block* that catches all possible exceptions. If the caught exception is an instance of the `HandshakeException`, the specified alert will be sent to the peer, otherwise the general alert `HandshakeFailure`.

5.6 Serialisation

Serialisation describes the process of converting an object (e.g., `Record` or `DTLS-Message`) into its raw binary representation to be sent over the network and later to be reassembled into an identical clone of the object at the peer's side. For this purpose, each object representing a *DTLS* message must have the following two methods:

`toByteArray()`

This method returns the byte representation of the specific object. It implements the protocol data structures defined in [10, 33].

`fromByteArray()`

This is the inverse function of the above one: It reads the received byte array stream and reconstructs the given object.

As mentioned, *(D)TLS* specifies data structures for each handshake message and how the serialisation is realised. Structure types are fundamentally constructed from primitive types and each specification declares a new, unique type. The syntax for definition is much like that of C:

```
struct {
  T1 f1;
  T2 f2;
  ...
  Tn fn;
} [[T]];
```

One of the important data structures is a *vector* which represents a stream of homogeneous data elements. There are two types of vectors:

Fixed-length

The syntax for specifying a new type `T'` that is a fixed-length vector of `T` is:
`T T'[n];`
Here, `T'` occupies n bytes, where n is a multiple of the size of `T`. Since, the length is fixed by definition, the length of the vector is not included in the encoded stream, unlike for the second type of vectors.

Variable-length

Variable-length vectors are defined by specifying a sub-range of legal lengths, inclusively, using the notation `<floor..ceiling>`. Since the length will be variable, the encoded stream of the vector must be prepended with a number consuming as many bytes as required to hold the specified maximum length of the vector. Otherwise, the decoding process would not work as it would be unclear how many elements the following vector contains. An example should illustrate this easily:
`uint16 longer<0..800>;`
This represents a vector of 0 up to at most 400 16-bit unsigned integers (800 is the maximum allowed length of the whole vector, not the maximum number of elements). First, we have to determine how many bytes are needed for the length field: The maximum length of the vector cannot be represented using just 1 byte (maximum length would be 255 for 1 byte), but 2 suffice (65535) to express the maximum length of the vector. This 2-byte length field is prepended to the actual vector. So, the vector `[1, 2, 3, 4]` would be encoded to:

```
0x00, 0x04, 0x01, 0x02, 0x03, 0x04
<-length->  <-------vector------->
```

Now, let us look at a concrete example on how serialisation is realised and what must be considered. We will use one of the most simple handshake messages, the *HelloVerifyRequest*: Listings 5.4, 5.5 and 5.6 show the definitions of the data structures which will be needed for this message.

Listing 5.4: The Record Layer data structure.

```
struct {
  ContentType type;
  ProtocolVersion version;
  uint16 epoch;
  uint48 sequence_number;
  uint16 length;
  opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;
```

Listing 5.5: The Handshake Protocol data structure.

```
struct {
  HandshakeType msg_type;
  uint24 length;
  uint16 message_seq;
  uint24 fragment_offset;
  uint24 fragment_length;
  select (HandshakeType) {
    case hello_request: HelloRequest;
    case client_hello: ClientHello;
    case hello_verify_request: HelloVerifyRequest;
    case server_hello: ServerHello;
    case certificate:Certificate;
    case server_key_exchange: ServerKeyExchange;
    case certificate_request: CertificateRequest;
    case server_hello_done: ServerHelloDone;
    case certificate_verify: CertificateVerify;
    case client_key_exchange: ClientKeyExchange;
    case finished: Finished;
  } body;
} Handshake;
```

Listing 5.6: The HelloVerifyRequest data structure.

```
struct {
  ProtocolVersion server_version;
  opaque cookie<0..2^8-1>;
} HelloVerifyRequest;
```

Figure 5.1 depicts the serialisation of the listed data structures. One can see that the transformation is quite straight forward, as almost all field definitions are clear. The only potential pitfall is the field `opaque cookie<0..2^8-1>` defined in the *HelloVerifyRequest* data structure. Since this field is a variable-length vector, we need to add a 1-byte length field in front of it. 1 byte suffices since the maximum specified length of `2^8-1` can be represented using 1 byte only.

So, in our implementation, serialisation is done by calling the `toByteArray()` method of the record that is responsible for the record header, which will then call the handshake message's `toByteArray()` function that will be responsible for the

handshaker header and the underlying handshake message. De-Serialisation is done in the same way by invoking the `fromByteArray()` method. Please note that the `toByteArray()` method is dynamically bound according to the type of the handshake message, while the `fromByteArray()` is a static method defined in the abstract class `HandshakeMessage` which calls the `fromByteArray()` method of the subclass according to the read `msg_type` field.

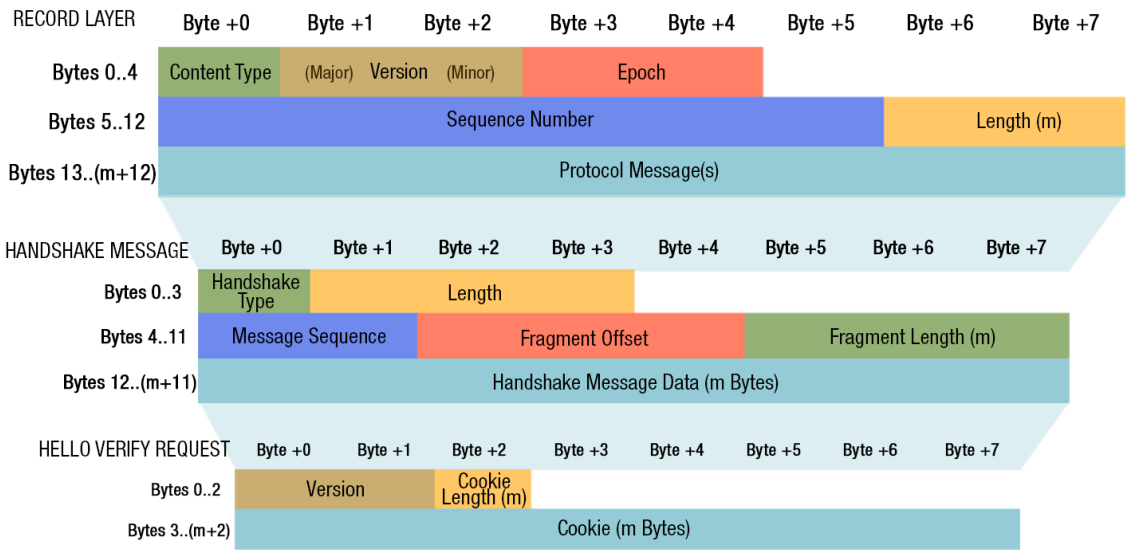


Figure 5.1: Serialisation of the *HelloVerifyRequest* with the handshake and record headers. The field `Content Type` will have the value 22 (standing for handshake message) and the field `Handshake Type` the value 3 (indicating a *HelloVerifyRequest* inside the handshake message data).

The challenge in implementing the serialisation lies in the diversity of the handshake messages: There are eleven handshake messages defined, several of them far more complex than the presented *HelloVerifyRequest*. Also, certain messages have different formats according to the used key exchange algorithm. Thus, implementing all variations, getting everything right as specified, and testing all the message formats was one of the greatest challenges faced during the implementation.

5.7 Fragmentation

Fragmentation has been specified in [33] to avoid *IP fragmentation*: Each *DTLS* message must fit within a single transport layer datagram. Since certain handshake messages are potentially bigger than the maximum record size (mostly X.509 *Certificate* messages), *DTLS* provides a mechanism for fragmenting a handshake message over a number of records. In order to support fragmentation, *DTLS* modified the *TLS* 1.2 handshake header by adding two new fields:

`fragment_offset`
: Indicates the number of bytes contained in the previous fragments. The first fragment starts with the `fragment_offset = 0`.

`fragment_length`
: Specifies the length of this fragment and must not be larger than the maximum handshake fragment size.

All fragments have the same `message_seq` value as the original message; the same is true for the `length` field in the handshake header. An unfragmented handshake message therefore has `fragment_offset = 0` and `fragment_length = length`.

To implement fragmentation, the class `FragmentedHandshakeMessage`, which extends the abstract class `HandshakeMessage` (see Figure 4.4), had to be created. It serves as storage data structure that contains one part of the fragmented message which is transparent to the object holding it. Before the handshake message is wrapped into a `Record` layer, it is checked whether the handshake message is larger than the maximum record size. If so, the message is divided into a series of N contiguous data ranges (hold by the `FragmentedHandshakeMessage`) which are then sent as N separate `Records` over the network. When a fragmented handshake is received—identified by the fact that `fragment_length` $\neq$ `length`—it is buffered until all fragments have arrived. Since all fragments of the same original message have the same `message_seq` value, they can be stored according to this value using a `Map`.

```
protected Map<Integer, List<FragmentedHandshakeMessage>>
                    fragmentedMessages;
```

Because of potential reordering of the handshake messages, each time a fragmented message has arrived, it must be checked whether all fragments have been received to reassemble the original message. Therefore, the list of `FragmentedHandshakeMessage` is ordered by the `fragment_offset` field and then iterated over to reassemble the original byte array. If the size of the reassembled byte array equals the size of the original message, the unfragmented `HandshakeMessage` object can be reconstructed by calling the corresponding `fromByteArray()` method. While iterating over the fragments ordered by the `fragment_offset`, one needs to take care of two special cases: a) The same fragment could be received multiple times, or b) the fragments could have overlapping ranges.

These two cases can be handled by remembering the next expected `fragment_offset` value, which is initially set to 0. A fragmented message can be used to reassemble the whole message when one of these cases is true:

1. *No overlapping*: The offset of the fragment is equal to the next expected offset.
2. *Overlapping*: The offset of the fragment is less than the next expected offset, but its offset in addition with its fragment length is greater than the expected offset. In this case, it is important to use only the bytes that do not overlap with the already reassembled part of the message.

If the message could be used, the next expected `fragment_offset` value is set to the current length of the reassembled part of the message.

5.8 Properties

The class `Properties` implements *Cf*'s property registry which is used to manage *CoAP*- and *Cf*-specific constants in a central place. The properties are initialised in the `init()` section and can be overridden by a user-defined `.properties` file. If the file does not exist upon initialisation, it will be created so that a valid configuration always exists. To make the use for *DTLS* easy configurable, the main security parameters defining the protocol have been added to this registry. These are the following (with the standard values in brackets):

USE_RAW_PUBLIC_KEY (`false`)
: Indicates whether the client wants to receive and potentially send its *Certificate* message containing only the public key or the full X.509 certificate chain. If the value is set to `true`, the client will send the `cert_type` in its *ClientHello* extension which will contain at first position the *raw public key* certificate type and at second the *X.509* indicating that it is capable of processing both *raw public key* and *X.509* certificates, but prefers to receive *raw public keys*. If the server supports *raw public keys* as well, its *ServerHello* message will contain the `cert_type` extension containing the *RawPublicKey* certificate type, otherwise the extension may be omitted [44].

CLIENT_AUTHENTICATION (`false`)
: This parameter defines whether the server requires the client to authenticate itself. This parameter is only decisive, if a ciphersuite has been selected that requires authentication (e.g., `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`). Thus, if this parameter is set to `true` and the server has sent its *Certificate* message, it sends a *CertificateRequest* message to the client and expects in the client's next flight an additional *Certificate* and *CertificateVerify* message; otherwise the handshake will be aborted.

KEY_STORE_LOCATION (`path/to/keyStore.jks`)
This gives the location where the keystore file is found. Normally, this should be an absolute path, unless the keystore is packed within the executable *Java Archive (JAR)* file. The keystore file is used to load the entity's private key and certificate chain (see 5.4 Certificates). Not providing any keystore file is acceptable, as long as a ciphersuite is chosen, where the certificate chain and private key is not needed (e.g., `TLS_PSK_WITH_AES_128_CCM_8`); otherwise it will result in a failed handshake.

TRUST_STORE_LOCATION (`path/to/trustStore.jks`)
Indicates the location of the truststore file (described in 5.4 Certificates). The truststore file contains all the trusted *CAs* certificates. If this file is not provided and the entity requires peer authentication, the certificate chain cannot be validated (unless self-signed certificates are accepted).

PREFERRED_CIPHER_SUITE (`TLS_PSK_WITH_AES_128_CCM_8`)
In the *ClientHello* message, the client provides a list of supported ciphersuites ordered by preference. Since the current implementation only supports two ciphersuites, this parameter suffices to configure which one is preferred. In future versions, when more ciphersuites will have been implemented, this should be made more flexible to configure the list more easily.

MAX_FRAGMENT_LENGTH (`200`)
This value defines the maximum length in bytes a handshake message fragment is allowed to have without being sliced into multiple fragments (see 5.7 Fragmentation for details). Please note, that this value indicates an upper limit for the `fragment_length` field in the handshake header, not the size of the total transport layer datagram.

RETRANSMISSION_TIMEOUT (1000)
The retransmission timeout is defined in milliseconds and indicates the amount of time which can elapse for the peer to successfully send the whole flight. After this, the current flight is retransmitted and the timer reset. The value of 1000 milliseconds is recommended in [33].

PSK_IDENTITY (`PSK_Identity`)
If a *PSK* key exchange algorithm is used to establish the *Master Secret*, the client must include in its *ClientKeyExchange* message an `psk_identity` indicating which pre-shared key to use. If this value is not given, the server will not find a pre-shared key which will lead to an aborted handshake.

5.9 Providing Reliability

The *TLS* handshake is a lockstep cryptographic handshake: Messages must be transmitted and received in a defined order—any other order is an error. Clearly, this is incompatible with the unreliable datagram traffic used in *DTLS*, namely reordering and message loss can happen. For both these problems, *DTLS* provides fixes.

5.9.1 Message Loss

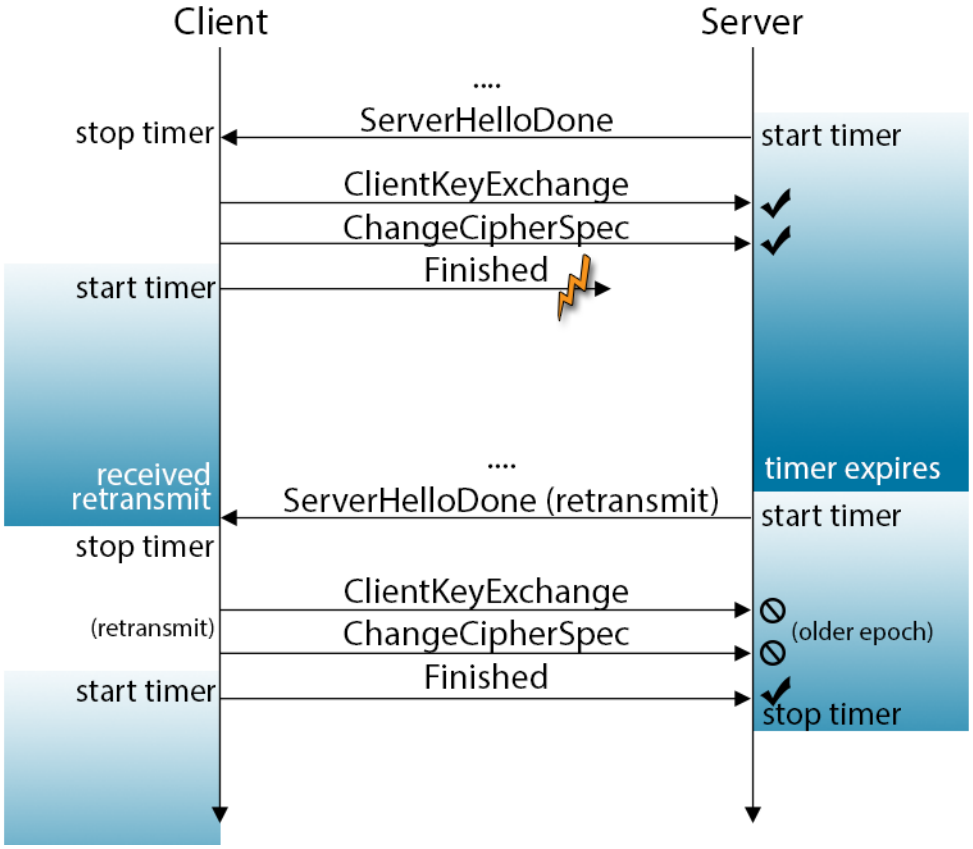


Figure 5.2: An example on how packet loss is handled: If the client's message is lost, the server will retransmit its last entire flight after the timer expires. The client will retransmit its entire flight as well when it receives a retransmit.

Figure 5.2 shows how message loss is handled in *DTLS*: It uses a simple retransmission timer to handle packet loss. After having sent the last message of the current flight, the server starts a retransmission timer. While the timer is running, the client received the server's whole flight (with the *ServerHelloDone* being the last message), therefore

cancelling its timer for its previous flight and preparing to send the next flight. The server receives both the *ClientKeyExchange* and *ChangeCipherSpec* message and expects now to see the client's *Finished* message. However, after the timer has expired and the message has not yet arrived, the server knows that the *Finished* message must be lost, because some messages of the client's current flight have already arrived. Therefore, it retransmits the whole flight to signal to the client that message loss has occurred. When the client receives the retransmissions, it knows it must retransmit. This message loss would also be handled by the client's timer, if for some reason the server would not retransmit in time.

Concretely, this problem is solved in our implementation with the classes `Timer` and `TimerTask`: Every time a new `DTLSFlight` is sent, the timer schedules a task that will be execute in the future (after the current timer value has expired). When the timer fires, it calls the `run()` method of the task that retransmits the current flight, adjusts the timeout and schedules a new retransmission task, if the maximum number of re-tries is not reached yet. When the last message of the peer's current flight is received, the current task is cancelled, the new flight is being sent, and a new retransmission task scheduled containing the current flight.

The timer value is a very delicate choice: A too small value can lead to serious congestion problems when each instance time out early and retransmit too quickly although no message loss has occurred. On the other hand, a too large value can cause really long round-trip times if message loss happened. The *DTLS* specification suggests an initial timer value of 1 second and to double the value at each retransmission, up to no less than the maximum of 60 seconds [33]. This is currently implemented in this way, but future tests in constrained environments will show whether this value needs adjustment.

5.9.2 Reordering

In an unreliable network, messages can arrive in a different order than they were sent out. A *TLS* implementation would break in such an environment since the order of processing the messages is crucial for the correctness of the protocol. Therefore, *DTLS* has made a few adjustments to the record and handshake header to handle reordering:

Epoch

The `epoch` field is a counter value which is incremented on every cipher state change; so, each time a *ChangeCipherSpec* message is sent. This field was introduced to the record header. It is used to distinguish from which encryption state the message comes: Only messages coming from the same epoch as the current read state can be understood correctly, as for each new epoch the write state of the sender will change.

Sequence Number

The `sequence_number` field is the current sequence number for this record. Sequence numbers are maintained separately for each epoch, with each sequence number initially being 0 for each epoch. Please note, that for each retransmit, the sequence number must be incremented.

Message Sequence Number

The `message_seq` field was introduced in the handshake header and keeps track of the handshake message counter during a handshake: Every time a new message is generated, the `message_seq` value is incremented by one. While the `sequence_number` must be incremented for each retransmit, it is important to note that the `message_seq` value stays the same for each retransmit.

So, how do these new fields help handling reordering? The concatenation of the `epoch` and `sequence_number` field make up an explicit sequence number, in contrast to *TLS* where such a sequence number can be established implicitly due to the reliable transport channel. If only reordering would happen, this explicit sequence number would suffice to handle it. But, in combination with message loss, we need the `message_seq` value as well. This is solved in our implementation with the `processMessageNext()` method in the `Handshaker` class. Whenever a new message is received, this method is called to determine what to do with the incoming message: Basically, there are three options: a) Process it right away, b) queue it for later processing, or c) discard it. To determine this correctly, we need two values: The current `epoch` which is stored in the `DTLSSession` associated with this handshake, and the `next_receive_seq` counter which is stored in the `Handshaker`. This counter is initially set to zero and keeps track of the next `message_seq` value which is expected to be processed. The `processMessageNext()` method works as follows: If the `epoch` value of the message, compared to the current `epoch`, is:

less

The message is discarded since it comes from an older epoch and therefore must be a retransmit.

greater

The message is queued for later processing. It must have arrived earlier due to message reordering. It is important to note that since the message comes from a future epoch, the current read state is not compatible with the write state the message was encrypted with. Therefore, the handshake message contained inside the record cannot be understood yet.

equal

The handshake message can be understood since the read and write states of the message are the same. We now need to examine the `message_seq` value in the handshake header.

If the `message_seq` matches the `next_receive_seq`, the counter is incremented and the message is processed. If it is less than `next_receive_seq`, the message must be discarded, but since it marks a retransmit, the whole flight must be sent again. If it is greater than `next_receive_seq`, the message is queued. Figure 5.3 illustrates the handling of the `sequence_number`, `message_seq`, and `next_receive_seq` counter. Please note that queuing for future messages is not defined to be mandatory in the *DTLS* specification [33]; the message may also be discarded. This is a space/bandwidth trade-off which might be interesting to examine for a constrained environment.

Queued messages are stored in a `Collection<Record>`. Every time a new message has arrived that could not be processed, or a message has been processed that did not trigger a new flight to be sent, the whole queue is scanned for the next needed message. If this message is available in the queue, it is removed and processed. Special caution must be taken when a fragmented handshake message is received. The `next_receive_seq` must only be incremented when all fragments of the messages have arrived, not beforehand.

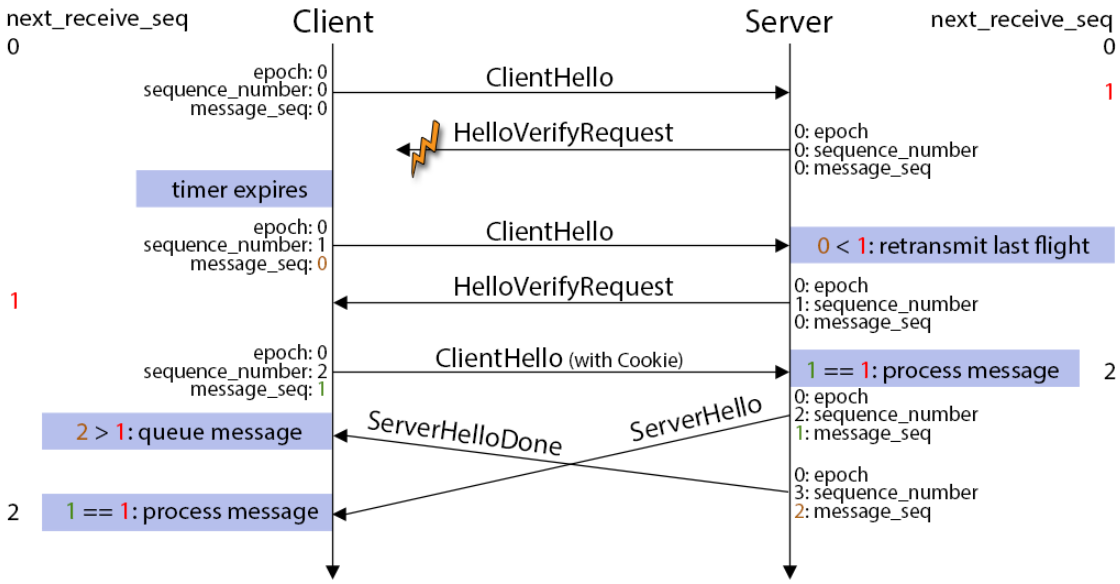


Figure 5.3: An example on how to handle reordering and retransmits: A message is only processed when the `epoch` and `message_seq` match the expected values; otherwise the message is either queued or discarded. Also note the value of the `sequence_number`: It increases with each retransmit while the value of the `message_seq` remains the same for each retransmit.

6 Analysis

6.1 Testing

The security of *DTLS* relies upon the security of *TLS* since it is designed to be as similar to *TLS* as possible. Therefore, mainly the security considerations of *TLS* need to be considered. Appendices D and E of [10] list possible implementation pitfalls and make recommendations for implementations to ensure security. Those have been followed very carefully to avoid attacks based upon false implementations of the specifications.

6.1.1 Interoperability Tests

Regarding the security considerations of *TLS* (Appendix F of the *TLS* 1.2 specification), an implementation is secure as long as the specification has been followed correctly [10]. To completely verify the conformation to the specification seems to be almost impossible, because the protocol is very complex and offers lot of pitfalls. Thus, we conducted various interoperability tests with other implementations which resulted in an increased confidence, that the specification has been implemented correctly. Table 6.1 shows what could have been tested:

	PSK	**ECDHE**	**Mutual Authentication**	**Fragmentation**	**Raw Public Keys**
SICS	✓	✓	✓		
TinyDTLS	✓				
Silicon Labs	✓	✓			

Table 6.1: Summary of the interoperability tests.

Swedish Institute of Computer Science

At the *Swedish Institute of Computer Science (SICS)*, Shahid Raza and Daniele Trabalza have been implementing *DTLS* in Java. Together, we conducted very exhaustive interoperability tests which were spread over a month of testing and fixing of our implementations. Lots of implementation problems could be identified and clarified together resulting in successful interactions for both mandatory ciphersuites `TLS_PSK_WITH_AES_128_CCM_8` and `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`. Additionally, mutual authentication with trusted certificate authorities has been successfully tested.

TinyDTLS

TinyDTLS is one of the first open-source implementations of the *DTLS* protocol for small devices [13] realised by Olaf Bergmann at TZI Uni Bremen[1]. Only the `TLS_PSK_WITH_AES_128_CCM_8` ciphersuite could be tested for interoperability with our implementation, since tinyDTLS lacks the support for any other ciphersuite and key exchange method.

Californium (Cf) Interop Server

An *interop server* has been set up at `http://vs0.inf.ethz.ch/` to give other implementations the possibility to test their product against ours. This is primarily done trying to expose flaws in either our or their implementation and test how our implementation handles faulty data.

Silicon Labs

After having set up the *interop server*, Silicon Labs[2] successfully tested their *DTLS* implementation, which is based on a well-tested *TLS* 1.2 implementation, against our server. As their source code is not publicly available, testing our client against their server implementation was not possible.

As can be seen in Table 6.1, two important features introduces by *DTLS* could not be tested with other implementations: *fragmentation* and *raw public keys*. This problem arose mainly due to unfortunate timing: Several *DTLS* implementations are currently in development, but not yet finished. The next deadline for mostly all of them will be the end of November 2012 when the next *ETSI/IPSO Plugtest*[3] will be held and *DTLS* implementations will be tested for the first time. Thus, these features will be tested after the end of this thesis. Yet, we are confident that the fragmentation works correctly although we could not test it with other implementations. Wireshark[4], a network protocol analyser that supports *DTLS* message parsing, is able to reassemble the fragmented messages properly. So, potential implementation errors like wrong fragment offsets can be ruled out. Unfortunately, Wireshark still lacks the support for *raw public keys*.

[1] http://tinydtls.sourceforge.net/
[2] http://www.silabs.com/
[3] http://www.ipso-alliance.org/nov-29-30-etsiipso-coap-plugtest-sophia-antipolis-france
[4] http://www.wireshark.org/

6.1.2 Fuzz Testing

The problem with interoperability tests is that the other implementations behave generally as expected. So, the implementation seems to work correctly as long as every step is executed as specified by the protocol. But like this, faulty data or wrong execution steps are not ruled out to compromise the implementation. To address this issue, we ran different fuzz tests where the implementation is fed invalid or random data at certain steps in the protocol. The goal of these test runs was to check that the server cannot be crashed or compromised. All of the tests ended with the server aborting the handshake by sending an *Alert* message to the peer containing a fatal handshake failure message. Also, in all cases, the server was able to continue operating normally after the test was run.

6.1.3 Unit Testing

DTLS is a very complex protocol which must be tested in its entirety to ensure its correctness. This is mainly done by the described *interoperability* tests. Still, certain individual parts of the implementation, which are crucial for the correctness, are tested in separate unit tests. We limit ourselves to the cryptographic features that have been implemented by us and assume that the used functionality of *JCA* has been implemented correctly. The unit tests cover the following functionalities:

- Pseudorandom Function,
- Hash-Based Message Authentication Code,
- Counter with CBC-MAC, and
- Serialisation of numeric data types.

The existing test cases[5] [22, 42] for these functionalities have all been passed.

6.2 Performance

6.2.1 Setup

Measuring the performance of the implementation and comparing it to other implementations would be an interesting reference point on how well our implementation operates. There are problems to realise this goal: a) By now, only a few *DTLS* implementations exist (most of them only support `TLS_PSK_WITH_AES_128_CCM_8`) and no standardised benchmarking has been realised so far. b) Since the *TLS* handshake is quite compute-intensive, the benchmark results highly depend on the used hardware, which makes

[5] http://www.ietf.org/mail-archive/web/tls/current/msg03416.html

comparisons with existing *TLS* results difficult. Thus, we decided to skip this featureless comparison and just focus on the different security modes and how much overhead *DTLS* creates compared to an unsecured *CoAP* connection. So, one round-trip consists of sending a *CoAP* POST request, establishing a secure connection first if necessary, and receiving the response from the remote endpoint. To get comparable results and exclude statistical outliers, 50 request have been sent to the server simultaneously. This should also help to analyse whether congestion takes place at the server side when a lot of requests arrive at the same time.

Figure 6.1 shows the mean round-trip time and standard deviation in milliseconds for the different security modes (from left to right):

NoSec
: *DTLS* is disabled, a unsecured *CoAP* connection is used.

PreSharedKey
: *DTLS* is enabled, `TLS_PSK_WITH_AES_128_CCM_8` is the used ciphersuite, and no authentication takes place.

Certificate I
: *DTLS* is enabled, `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` is the used ciphersuite, *no* client authentication, and *no* fragmentation of handshake messages.

Certificate II
: *DTLS* is enabled, `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` is the used ciphersuite, *with* client authentication, and *no* fragmentation of handshake messages.

Certificate III
: *DTLS* is enabled, `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` is the used ciphersuite, *with* client authentication, and *with* fragmentation of handshake messages (maximum `fragment_length` = 100).

CoAPs
: *DTLS* is enabled, but a handshake has taken place prior to this request. So, an active session is already available and no handshake must be executed.

One remark about the different certificate modes: Tests have been conducted for both the full X.509 Certificate and the *raw public key* mode and they showed that using *raw public keys* has no positive effect on the round-trip time (but of course on the total transferred data). Since these tests were conducted in an unconstrained environment, network latency and throughput seems not to be a decisive factor in the overall performance of the system and therefore the use of *raw public keys* does not yield any performance improvement. Thus, for the sake of clear presentation, the *raw public key* data was dropped in Figure 6.1.

For the *CoAPs* mode, only one measurement is displayed because the round-trip time does not depend anymore on the used key exchange algorithm, but only on the chosen block cipher, which is the same in all security modes.

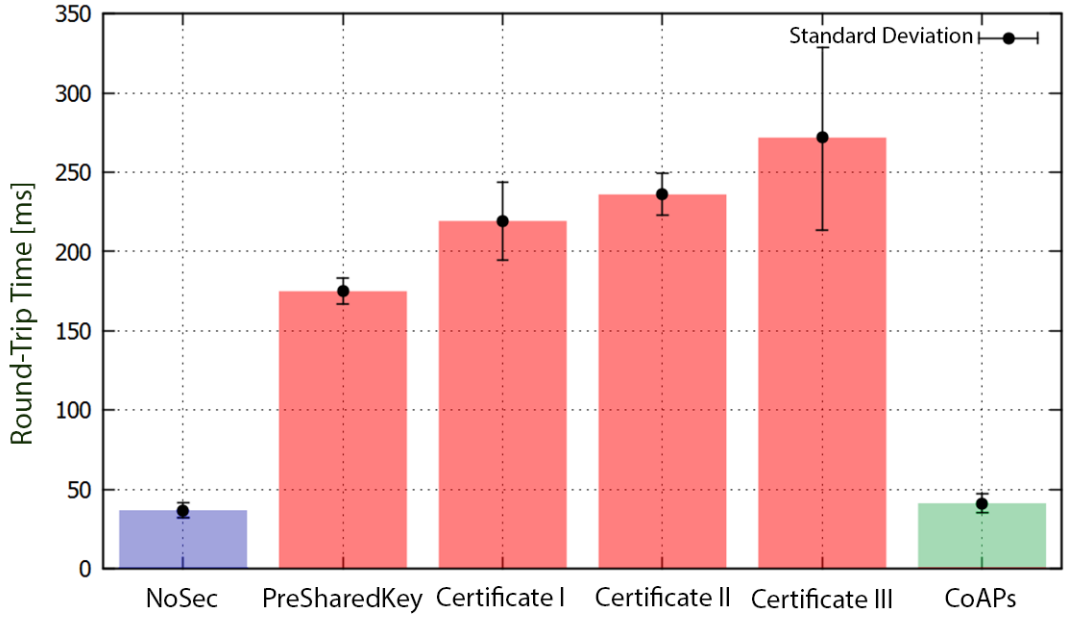


Figure 6.1: Comparison of the mean round-trip times for different security modes. The black error bars depict the standard deviation, illustrating the variation of the single measurements.

6.2.2 Discussion

Figure 6.1 clearly shows the immense overhead created when establishing a secure connection: *DTLS* adds at least (in this setup) 140ms to the round-trip time compared to an unsecured connection. Table 6.2 shows where one part of this overhead comes from: a) While an unsecured *CoAP* connection only needs 2 flights to complete the request (1 for the request plus 1 for the response), a secured connection needs 8 flights (6 for the handshake plus 2 for the encrypted response and request). These additional flights take a certain time on the network resulting in a longer round-trip time. b) The *DTLS* handshake adds a lot of payload which needs to be transferred and parsed. This becomes especially large when using mutual authentication and fragmentation: The proportion of the header data increases drastically when the fragmentation size decreases and can become up to 55% of the total number of bytes transferred [15].

The other part of the security overhead comes from the fact that the compute-intense operations of the handshake need a significant amount of time, i.e., the key agreement and data verification. The impact of the key agreement protocol on the round-trip time can

	Total #Flights	Total #Datagrams	Total #Bytes	Time [ms]
NoSec	2	2	122	36.72
PreSharedKey	8	8	1008	174.94
Certificate I	8	8	1791	219.21
Certificate II	8	8	2602	235.97
Certificate III	8	27	3727	271.86
CoAPs	2	2	174	41.12

Table 6.2: Summary of the performance test.

be seen when comparing the times of the *PreSharedKey* mode with the *Certificate* mode: The *PreSharedKey* mode does not need to run the key agreement protocol since it also has a pre-shared key which can be used, and therefore finished significantly faster than the *Certificate* mode. As can be seen by the large standard deviation for the *Certificate III* mode, fragmentation has an impact on the server's performance causing congestion to happen. This can be explained with the fact that fragmentation increases the number of sent flights drastically: 27 flights, compared to the 8 flights needed, cause more processing (e.g., reassembly of fragments, sending of each datagram individually, handling reordering or retransmissions) at both client and server side and therefore keeps them more utilised causing longer waiting periods for certain flights.

By comparing the *NoSec* with the *CoAPs* mode, we can estimate the overhead created when protecting a *CoAP* message: The mean round-trip time of a protected message is only marginally larger than an unprotected one and amounts to approximately 5ms in this setup. With this result we can argue that adding security to a communication connection is not costly once an active session has been established. Therefore, a session should be kept alive as long as possible to avoid another expensive handshake.

6.3 Profiling

In this section we take a closer look at our implementation *CPU*- and memory-wise. This analysis should provide us with a baseline for future optimisations. As the profiler we used Java VisualVM[6].

[6]http://visualvm.java.net/

6.3.1 CPU

We monitored a *DTLS* server instance for a longer period of time during which many client requests have been handled. Since the server is idle most of the time (the `DatagramSocket` is waiting to receive a datagram), we focus our analysis on the *CPU* times when the server is busy. For the handshake the `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` ciphersuite was negotiated with mutual authentication and fragmentation to identify all potential bottlenecks. Table 6.3 shows the hottest method when the server is executing the *DTLS* handshake protocol and application data is sent. We can notice that the use of *ECC* and authentication holds a large amount of the used *CPU* time. Therefore, we monitored a server instance when the `TLS_PSK_WITH_AES_128_CCM_8` ciphersuite was negotiated to compare them later in the discussion section. The results are listed in Table 6.4. A few remarks must be made about these numbers: We tried to profile our implementation with two other profilers as well, namely *YourKit*[7] and *Eclipse Test & Performance Tools Platform Project*[8]. The results have been quite diverse, especially when comparing the total used *CPU* times during the same period of time. But nonetheless, in all three results we could observe a similar trend towards the same methods that seem to be using a lot of *CPU* time. These experiments have been executed on a quad-core processor with 4 gigabyte of RAM.

Method	**Time [ms]**	**%**
`ECDHServerKeyExchange.<init>`	338	13.5
`CertificateVerify.verifySignature()`	313	12.5
`ECDHECryptography.<init>`	288	11.5
`EndpointAddress.toString()`	224	9.0
`ECDHECryptography.getSecret()`	207	8.3
`Handshaker.loadKeyStore()`	118	4.7
`CCMBlockCipher.decrypt()`	117	4.7
`Handshaker.loadTrustedCertificates()`	106	4.2
`DTLSLayer.sendFlight()`	65	2.6
`CCMBlockCipher.encrypt()`	54	2.2
`Log.format()`	52	2.1
`DatagramWriter.writeBytes()`	45	1.8
`Handshaker.wrapMessage()`	43	1.7
`DatagramReader.readBytes()`	39	1.6

Table 6.3: *CPU* usage by methods when using ECDH as the key agreement protocol.

[7] http://www.yourkit.com/
[8] http://www.eclipse.org/tptp/

Method	Time [ms]	%
`CCMBlockCipher.decrypt()`	99	16.8
`Handshaker.loadKeyStore()`	98	16.7
`Log.format()`	48	8.1
`DTLSLayer.datagramReceived()`	39	6.7
`EndpointAddress.toString()`	35	6.0
`DatagramWriter.writeBytes()`	19	3.3
`CCMBlockCipher.encrypt()`	19	3.3

Table 6.4: *CPU* usage by methods when using a pre-shared key. Please note that this table has been cut and only shows the methods that used more than 3% of the *CPU* time which is why the numbers will not add up to 100%.

6.3.2 Memory

We studied the memory usage over a longer period of time with "normal" usage of the server: We consider normal to be 10 requests per minute, three of them almost simultaneously. Figure 6.2 shows the available and used memory over a five minute period. We also examined the memory consumption of important classes during the handshake protocol. Table 6.5 shows the size of the objects after a certain number of successful handshakes (using `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`) and Table 6.6 the sizes after certain number of aborted requests: The client drops its last handshake message (the *Finished* message) which causes the server to wait for this message without completing the handshake. Please note that during this waiting period, the server keeps retransmitting its last flight, as the client's flight has not arrived in its entirety. When the maximum number of retries is reached, the server will not retransmit its flight anymore. But, as we are currently operating in an unconstrained environment, the associated handshaker and session remain stored (as the *DTLS* specification does not clearly state what to do in that case). In a constrained environment, one needs to be more cautious when allocating memory and not freeing it when the handshake seems to have failed.

Two remarks about the measurements: a) The "retained size" is calculated by the amount of memory allocated to store the object itself, not taking into account the referenced objects, plus the shallow sizes of all the objects that are accessible, directly or indirectly, *only* from this object. In other words, the retained size represents the amount of memory that will be freed by the garbage collector when this object is collected. b) By the given definition, it must be noted that the size of the stored `DTLSSessions` and `Handshakers` are contained in the total size of the `DTLSLayer`, since they are stored and accessible only by the `DTLSLayer`.

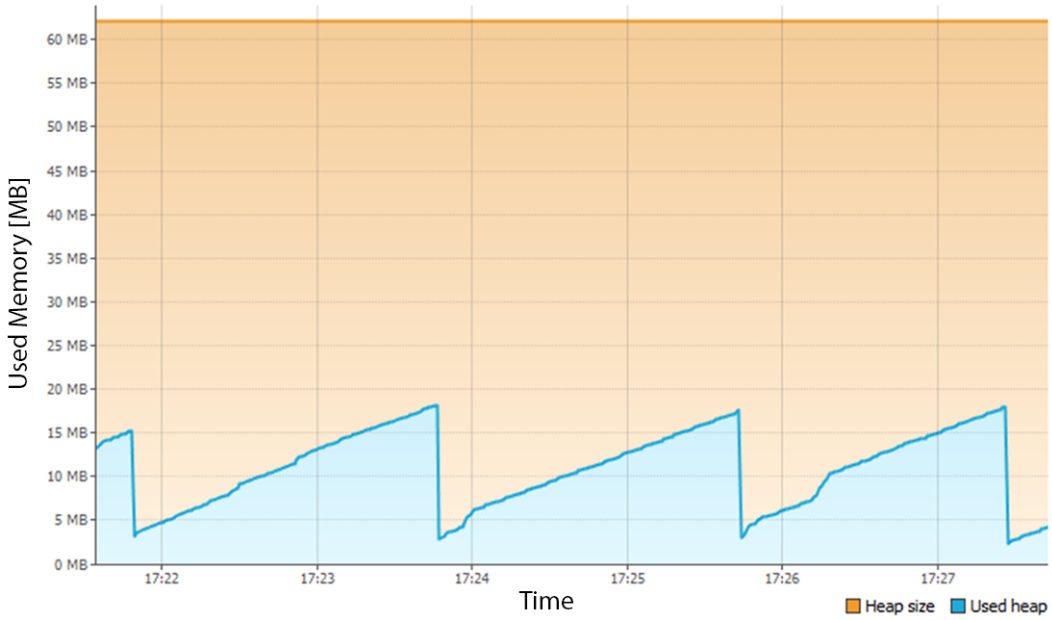


Figure 6.2: The memory usage over a five-minute period. The peaks mark the moments before garbage collection occurs.

	Initial	1 Request	2 Requests	5 Requests
DTLSSession	0	664	1328	3320
Handshaker	0	0	0	0
DTLSLayer	512	1304	2080	4408

Table 6.5: Retained size of objects (in bytes) after a certain number of successful requests.

6.3.3 Discussion

Let us first examine the *CPU* measurements: The data in Table 6.3 shows what could have been expected from the beginning: Using *ECC* for the key agreement protocol is very resource-intensive and uses a lot of the *CPU* time. Additional, loading the certificate chain and private key, and parsing the peer's certificate chain and verifying it takes up a lot of *CPU* time as well. Thus, one of the conclusions we can draw from this data, regarding the use of this protocol in constrained environments, is that we should use whenever acceptable a *PSK* algorithm for key agreement, or an out-of-band public key validation, or both. The data in Table 6.4 confirms this assumption. When a pre-shared key is used to calculate the *Master Secret*, the encryption is the defining factor for the *CPU* time, which seems acceptable as it is the main security feature.

Another detection for future optimisation is that resources should be loaded lazily. The `loadKeyStore()` method, which loads the certificate chain and the entity's private key, is invoked at the initialisation of the handshaker but never used during the handshake

	Initial	1 Request	2 Requests	5 Requests
DTLSSession	0	360	720	1800
Handshaker	0	6104	12208	30520
DTLSLayer	512	7352	14176	34648

Table 6.6: Retained size of objects (in bytes) after a certain number of aborted requests.

protocol because the `TLS_PSK_WITH_AES_128_CCM_8` ciphersuite does not require any authentication. In an unconstrained environment, this is not really a consequence, but for constrained environments this could be a crucial optimisation. Another remark about the different *CPU* times of the `encrypt()` and `decrypt()` methods: This data could somehow imply that the `decrypt()` methods uses more *CPU* time or is invoked more often, but this is not the case. Both methods should use about the same *CPU* time, but since the server must first decrypt the client's finished message, and only afterwards encrypts its message, the first-time loading of the underlying `Cipher` instance must be the limiting factor.

Now, let us go over the memory measurements. Figure 6.2 depicts a normal behaviour of such applications: The used memory increases linearly until the garbage collector executes another garbage collection round which brings the used memory back to the normal level. So, during normal execution no memory leak is apparent. This can be seen in Table 6.5: After a successful client request, the memory needed by the `DTLSLayer` increases roughly by the size of the `DTLSSession` which is stored while the connection is still active or resumable. The used `Handshaker` is deleted after finishing the handshake and does not consume any memory. So far, no mechanism has been implemented that removes old sessions to free memory although they could still be used. Though, with a size of approximately 700 bytes, this does not seem to be a problem in an unconstrained environment.

On the other hand, Table 6.6 shows a potential weak point of our implementation in an unconstrained environment but especially in a constrained environment: During these measurements, the client did not send its last handshake message causing the server to wait for this message indefinitely. Thus, the handshake cannot be completed and the associated `Handshaker` remains stored in the `DTLSLayer`. So far, there is no mechanism to delete such failed `Handshakers` after a certain period of time, therefore the server's available memory could be overloaded with such an attack. For future implementations, this must be handled to prevent nodes with constrained memory from becoming vulnerable to such attacks.

7 Discussion

In this chapter, we will summarise the findings of the last chapters, state the current status of our implementation, what has been achieved and what still needs to be done. Further, we will deliver insight into the problems we had during the implementation and testing phase.

7.1 Threat Model

In Chapter 3 we introduced nine potential security threats which must be considered when trying to provide security for the *IoT*. Since *DTLS* is a security protocol that operates at the application layer, four security threats located above this layer cannot be handled solely by *DTLS* and remain a potential security risk: *device cloning*, *substitution*, *privacy threat* and *extraction of security parameters*. As mentioned, this part of physical security threats remain a problem that still needs reasonable countermeasures coupled with *DTLS*. So, what about the remaining five security threats?

Firmware Replacement
A thing secured exclusively by *DTLS* is not protected against *firmware replacement attacks*. If an attacker is able to exploit a firmware upgrade to influence the operational behaviour of a thing, there is no use for encrypted communication and authentication since the thing has been compromised at a higher level rendering this endeavour useless.

Eavesdropping
DTLS provides full protection against *eavesdropping*. This is true for our implementation as long as the pre-shared key is not compromised and as long as the Diffie-Hellman assumption holds (which seems not to be at risk at the moment). An eavesdropper who does not know the private Diffie-Hellman values or the pre-shared secret has no option to derive the *Premaster Secret* and therefore is unable to decrypt the protected application data.

Man-In-the-Middle Attack
Whenever the server is successfully authenticated, the channel is secure against man-in-the-middle attacks. Only completely anonymous sessions are inherently vulnerable to such attacks; but we do not negotiate ciphersuites that support anony-

mous sessions. If the server is authenticated using a public key infrastructure or an out-of-band mechanism, an attacker has no possibility to forge the server's *ServerKeyExchange* message which is signed with the appropriate private key. When a *PSK* key exchange algorithm is used that does not require the server to authenticate itself, the authentication has already taken place before the actual handshake occurs: By knowing the pre-shared key, which must have been distributed in a secure and confidential way, both parties can be assured that there is no man-in-the-middle attack going on because an attacker would fail to forge the authentication tag in the cipher.

Denial-of-Service Attack

DTLS is susceptible to a number of *DoS* attacks. While the introduction of stateless cookie exchange in the server's *HelloVerifyRequest* message prevents *DoS* amplification attacks, and therefore protects constrained nodes, the server still remains a vulnerable target. Although the cookie exchange protects the server from executing *CPU*-intensive computations, a large number of simultaneously initiated sessions can still cause the server to become unavailable for its intended users.

Routing Attacks

Routing attacks take place at the network layer where an attacker can manipulate routes and the traffic over certain nodes. *DTLS* has no control over the path the packets take in the network. So, *DTLS* is vulnerable to certain routing attacks like *sinkhole* and *selective forwarding* attacks, but is secure against spoofing or altering of messages.

As we have noted before, *DTLS* alone will not prevent *node capture* attacks where an attacker captures a physically unprotected thing and tries to extract security parameters. So, what is the worst that could happen to a node secured with the current *DTLS* implementation exposed to such an attack? Let us examine what kind sensitive data is stored on a single node:

Certificates and Private Key

The certificate chain and corresponding private key must be stored on the node to be easy accessible by *DTLS*. For *Cf* we used a password-protected Java keystore, but any other data structure is possible. Even if the certificates and keys are password-protected, preventing an intruder from extracting those is hard since the password itself must be stored on the node itself. So, if the flash memory of a node can be accessed, the password and therefore the certificates and keys can be extracted [16]. The only protection against such attacks is some sort of certificate revocation list to invalidate the certificate once it has been compromised [5].

Pre-Shared Keys

The pre-shared keys are currently stored directly in the Java code. So, if the attacker gets hold of the executable *JAR* file, he can easily extracts those values, compromising all connections where these keys will be or have been used. So, if an

attacker is able to extract the pre-shared keys and if he has been eavesdropping past handshakes, he will be able to decrypt all encrypted messages afterwards; *PSK* does not provide *Perfect Forward Secrecy (PFS)* [12]. This is because one needs only the pre-shared key and the client's and server's random values (which are transmitted unsecured) to generate the *Master Secret*. Therefore, protecting the pre-shared keys from being extracted should be a high priority issue.

Sessions

During a lifetime of a node, it will establish secured connections to various peers. Each time a handshake was successful, the node stores a session containing two sensitive values: the *Master Secret* and the session identifier. Having obtained these values, an attacker can initiate future secured connections and impersonate this node using these two values in a resuming handshake. Additionally, since the *Master Secret* is compromised, all old conversations secured with it, can be decrypted. How easy it would be for an attacker to extract these values out of the memory of the running program is hard to tell [5], but as it poses a potential security risk, this should be examined further in the future.

As we have seen, using a *PSK* key exchange algorithm has its advantages, especially a simplified and faster handshake, but it is also a potential threat to security once a key has been compromised as it does not provide *PFS*. In contrast, using ephemeral keys as in *ECDH* key exchange guarantees *PFS* (as long as the *Master Secret* remains secured). Thus, for a more complex and time-consuming handshake, we gain this security feature.

7.2 Performance Optimisation

In Section 7.5 we gave several ideas for simple performance optimisations regarding memory and *CPU* usage. And, as usual in software engineering, there would be lot of additional possibilities to achieve better performance. But, at the current stage of the implementation, making this a high priority issue would be the wrong decision: The implementation has currently several untested features which makes it even harder to refactor the current code for optimisation reasons. Also, especially in a security system, a correct implementation should always be favoured over a fast (and maybe faulty) implementation. Currently, due to the lack of other implementations, we are too short on experience about the stability and correctness of the implementation. Once we have gained more insight about this, optimisation can be considered. Additionally, *Cf* will undergo refactoring in the future which will result in necessary changes to the *DTLS* implementation for integration purposes. This must also be solved prior to optimisation.

7.3 Known Issues

In this section, we list the known issues with our current implementation. Some of them can be seen as optional optimisations and others as required fixes as they contradict with the specification; these have mainly been detected during the writing process while rereading the specifications, but not during the testing phase. First, the optional changes:

- The `DTLSLayer` should be able to handle both secured and unsecured *CoAP* messages. Currently, if the `DTLSLayer` receives an *UDP* datagram, it interprets it as a *DTLS* record and parses it accordingly to the *DTLS* message format. So, if an unsecured *CoAP* message is received, this will result in an error due to the different message formats. This is not really a problem, since the *CoAP* specification defines the `coaps` scheme and a separate port for secured connections. Thus, if a plain *CoAP* is received at this port, it is acceptable to discard it (because it should not be sent to this port), but it would be a nice-to-have feature to handle this case as well.

- When a certificate chain is validated, the trusted certificates configured in the `cacerts` keystore file should be used as well. So far, we only load the truststore file, which is configured in the `Properties` class, containing certain trusted certificates of *CAs*. But the Java runtime environment contains a system-wide keystore `cacerts` with *CA* certificates which should also be utilised.

- When deployed in a real constrained network, self-signed certificates should not be allowed anymore. For testing purposes—not everybody has a certificate signed by a commercial *CA*—we allow a peer to authenticate itself using a self-signed certificate. Self-signed certificates can easily be manipulated and must not be used for real-life authentication.

- The cookie exchange during the handshake should be performed by the server without generating any state. The stateless cookie exchange has been introduced to prevent *DoS* attacks. Currently, the server allocates already a new `Handshaker` and `DTLSSession` object when a *ClientHello* messages is received for the first time. But this should not be done until the client has replied with the correct cookie to protect itself from *DoS* attacks which try to consume as much memory of the server as possible.

- To support as many named curves as possible when using *ECDH* key agreement, all recommended elliptic curves parameters should be implemented: There are 25 named curves specified in [6]. Currently, only `secp521r1`, `secp384r1`, `secp256r1`, `secp256k1`, `secp224r1` and `secp224k1` are supported.

- Change the 2-byte identifier of the ciphersuite `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` once it is officially specified by *IANA* [40].

- *CoAP* defines two mandatory to implement ciphersuites which both are supported in the current implementation. While the focus of other *DTLS* implementations also lies on these two ciphersuites, it would be advisable to implement others as well. Especially, because so far only *AEAD* ciphers have been implemented while block ciphers are still missing.

Following, changes that need to be implemented to conform to the protocol specification:

- When the server sends its ephemeral *ECDH* public key, all types of elliptic curve parameters must be supported. Currently, only the use of named curves is implemented and although recommended by the specification, the two other types must also be supported: `explicit_prime` and `explicit_char2`.

- Whenever a *ChangeCipherSpec* message is received, it must be guaranteed that all prior handshake messages have been received. Right now, when a *ChangeCipherSpec* message is received, it is processed and the current read state is updated. But due to message lost and reordering, it is possible that certain messages have not yet arrived and would be interpreted with the wrong read state. Of course, this can only happen, when the peer's flight does not fit into one datagram.

- The Internet-Draft [44] introduces the support for *raw public keys* in *DTLS*. While the support for sending *raw public keys* and the handshake extension `cert_type` is implemented, an out-of-band public key validation mechanism, like *DNS-based Authentication of Named Entities (DANE)* [17] or pre-configured keys, is still missing.

- Multiple handshake messages may be placed in the same *DTLS* record, provided that there is room and that they are part of the same flight. Thus, there are two acceptable ways to pack two *DTLS* messages into the same datagram: in the same record or in separate records. Currently, only one way is supported: Each handshake message is put into a separate record. During the implementation phase, the specification was misinterpreted and this error was not detected during the testing phase either. What needs to be changed is the parsing of the record layer while the sending can remain unchanged since it is not required that both ways must be supported for the sending part (only for the receiving).

- *DTLS* records contain a sequence number to provide replay protection. Sequence number verification can be performed using a sliding window procedure with a windows size of 32 or 64. The specification defines this feature to be recommended but not mandatory, nonetheless it should be implemented as soon as possible.

7.4 Implementation Notes

In this section, we want to share a few advices and lessons we learnt during the implementation and especially during the testing phase. They mainly consist of interpretation problems of the different specifications.

- The first *ClientHello* (not containing the cookie), *HelloVerifyRequest*, and *ChangeCipherSpec* messages are not included in the handshake messages hash used in the `verify_data` of the *Finished* message; the same is true for the handshake messages used in the *CertificateVerify* message.
- The second *Finished* message must also contain the first *Finished* message in the handshake hash to compute the `verify_data`.
- Although the `verify_data` can have a variable length in the *Finished* message, there is no length field prepended to it during the serialisation. The length is defined by the used ciphersuite and is normally 12 octets.
- The Finished MAC must be computed as if each handshake message had been received in a single fragment.
- There are several hello extension types, so it is possible that not each one is supported by an implementation. As long as a ciphersuite is negotiated that does not need this specific extension, it is no problem to continue the handshake. But, one needs to make sure that the unknown extension is still included in the handshake messages hash in the *Finished* message, otherwise the `verify_data` will not match, resulting in an aborted handshake.
- Although *DTLS* modified the *ClientHello* message format in such a way that the `extensions` field was dropped, the client must be able to send extensions in its hello message; this field must have been forgotten.
- A variable-length vector in the message format definition must always be preceded with the actual length when encoded, even if the length is clear from the given context. For example, when receiving the *ServerHello* message containing the certificate type extension, we know that the server's supported certificate types contains exactly one element; nevertheless we need to add a length field in front of it, since the extension data is defined as a variable-length vector:

  ```
  opaque extension_data<0..2^16-1>;
  ```

- When using an *AEAD* cipher, we need to generate additional authentication data, which is defined as follows:

  ```
  additional_data = seq_num + TLSCompressed.type +
                    TLSCompressed.version + TLSCompressed.length;
  ```

The `seq_num` is defined for *DTLS* as the concatenation of the 16-bit epoch and the 48-bit sequence number of the record header. Additional, the explicit nonce, which must be prepended to the ciphertext, has the same value.

- The node that transmits the last flight of the handshake (the server in an ordinary handshake or the client in a resumed handshake) must respond to a retransmit of the peer's last flight with a retransmit of the last flight. But it must not retransmit its last flight because of an expired timer; there is no timer set in this state of the handshake.
- When a connection is established by resuming a session, new `ClientHello.random` and `ServerHello.random` values are hashed with the master secret of the session which has been generated in the first full handshake.
- Whenever a digitally-signed element is encoded, the used signature and hash algorithm must be included as well. This is the case for the *CertificateVerify* message.
- The security parameter values for the ciphersuites `TLS_PSK_WITH_AES_128_CCM_8` and `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` are:
  - PRFAlgorithm: tls_prf_sha256
  - BulkCipherAlgorithm: aes
  - CipherType: aead
  - EncKeyLength: 16 octets
  - BlockLength: 16 octets (not relevant for *AEAD* ciphers)
  - FixedIvLength: 4 octets
  - RecordIvLength: 8 octets
  - MACAlgorithm: null
  - MACLength: 0 octets
  - MACKeyLength: 0 octets (no MAC because of the use of *AEAD* ciphers)
  - CompressionMethod: null

7.5 Optimising the Current DTLS Implementation

Our current implementation of *DTLS* for *Cf* was not specialised for use in constrained environments—and probably will never be since it is implemented in Java. Still, we would like to point out a few potential implementation optimisations.

7.5.1 Memory Usage

In our implementation, we utilise several `Maps` and `Collections` to store associated information (e.g., `DTLSSessions`) or to provide buffering (e.g., for `HandshakeMessages`). These data structures grow according to the number of elements until the available memory is used up. This can be very dangerous for constrained nodes because it makes them vulnerable to *DoS* attacks that try to exhaust the victim's resources. One reasonable optimisation would be to allow only a fixed number of slots for each buffer instead of dynamically allocating more memory whenever new items must be buffered. This approach would guarantee an upper bound for used memory, but at the same time it creates the risk of limiting *availability*: Requests may not be processed due to missing slots. Furthermore, an algorithm that removes old `DTLSSessions` could help saving memory. But it is important to note that deleting a `DTLSSession` forces the parties to execute the full handshake instead of the abbreviated handshake to establish a new secure connection. The trade-off between low memory usage and *CPU*-intensive operations must be evaluated carefully.

7.5.2 CPU Usage

The *CPU* usage is mainly bound by the cryptographic operations that must be executed during the handshake. Whenever possible, as described, the implementation utilises the built-in functionality of Java. Since it is widely used and tested, performance should be hard to be optimised: A handmade implementation may perform slightly better than one of Java, but this introduces the risk of bugs due to a faulty implementation which is not worth taking for a potential slight optimisation. One optimisation could be realised when negotiating the ciphersuite. The server should not choose the first supported ciphersuite proposed by the client, but the one which has the least *CPU*-consuming operations while guaranteeing the required security level. So, for the current implementation, the server should always chose the `TLS_PSK_WITH_AES_128_CCM_8` over the `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` ciphersuite—as long as the client proposes both—disregarding the client's preferences.

7.6 Future Implementations

As we have mentioned, the current implementation is not suited for constrained environments. Even if we would optimise the implementation and use a *JVM* designed specifically for resource-constrained nodes [2], the memory- and code-footprint will probably still be too large. Therefore, the next step will be to implement *DTLS* in a low-level programming language, most likely in C. During the implementation of *DTLS* for *Cf*, we gained experience about the complexity of the different parts that make up the *DTLS* protocol and we evaluate them on their potential impact on a low-level implementation:

Serialisation
Although quite complex in the understanding, serialisation should not be difficult and costly to be implemented. Especially, since the handling of low-level data structures such as byte arrays is more intuitive and simpler than in Java, and the specified message formats are already in a C-like structure.

DTLS State Machine
The *DTLS* state machine, which controls the handshake message flow and handles timeouts and retransmissions (in *Cf* called the *DTLSLayer*), is one of the core pieces in *DTLS*. As it must provide lots of different functionality, its code-footprint will potentially be one of the largest in the implementation.

Encryption
The mandatory *AES-CCM* encryption algorithm has been implemented by us in Java. This code could be ported to C, resulting already in an acceptable code-footprint. Also, there exist several implementations for the efficient use in resource-constrained environments [18, 34].

Fragmentation & Retransmissions
As we have discussed in Section 3.5, *CoAP* could be used to transfer the *DTLS* handshake messages instead of the *DTLS* protocol. Like this, support for fragmentation and retransmission could be left out, as *CoAP* provides this already. In *Cf*, the implementation of fragmentation is only about 200 lines of code, which will look similar in the C code. So, this is not a strong argument in favour of using *CoAP* to transfer the *DTLS* messages. For retransmissions this looks a little bit different: Implementing retransmission in *DTLS* requires primarily a data structure that holds the whole flight that needs to be retransmissioned once the timer expires. Therefore, memory-wise it would be wise to let *CoAP* handle this directly, instead of implementing a very similar mechanism again for *DTLS*.

Let us also take a look at some estimations about the needed buffer sizes to successfully execute a handshake: Due to reordering, we need buffers to store handshake messages, to process them in the right order and to compute the handshake messages hash correctly (used in the *Finished* message). So, we cannot handle everything on-the-fly. In a worst-case scenario, the client needs to buffer around 1 kB of handshake messages data: This can

happen, when mutual authentication is required and the *ServerHello* message is lost. Until the message is retransmitted, the client must store the *Certificate* (containing the full X.509 certificate), *ServerKeyExchange*, and *CertificateRequest* messages. At the same time, the client must also provide buffers for the messages that might need to be retransmitted, and store intermediary results that are needed at a later point in the handshake (e.g., the random values, the digest of the handshake messages or the server's public and ephemeral public key). So, roughly, 2-3 kB of data must be stored during a handshake.

This could be reduced when reordering is handled in another way: When we always discard a received message if it is not the next one expected, we do not need any buffers to store the handshake messages that arrived out of order. Additionally, for retransmissions we do not need to store the whole flight, but just the important information, and re-generate the flight each time the timer expires. So, by only storing the intermediary results and not providing any buffers, we can largely reduce the expected amount of data to be stored during a handshake at the cost of a potential higher handshake duration (due to the increased number of retransmissions) and higher *CPU* usage.

This changes again, when fragmentation must be handled. In this case, we must provide at least some buffers: Even if we still discard each message that arrives out of order, we need to store the part of the fragmented handshake message that has already been received until all fragments have arrived because parsing the fragments on-the-fly would only complicate the problem: For example, parsing the first 200 bytes of a X.509 certificate would be of no advantage, as the parsed part (and the parts that could not yet be parsed completely) must be stored somewhere as well, and reassembling parts of a certificate is far more complicated than parsing it all at once. Thus, as a certificate can be about 1 kB in size, we must at least be able to buffer this size of fragmented handshake messages.

8 Conclusion

The goal of this thesis was to evaluate the planned security model for the *Constrained Application Protocol (CoAP)*. We gave an overview of the security threats that exist during the different phases of the lifetime of a thing (i.e., manufacturing, installation and commissioning, and operational) and outlined the measurements to ensure security during these phases.

We presented a roadmap that outlines ideas to make *Datagram Transport Layer Security (DTLS)* more friendly for constrained environments. Among other ideas, Hartke & Bergmann propose a stateless header compression algorithm that can help reducing the header size of *DTLS* messages considerably. This approach looks very promising as the *DTLS* headers produce unnecessary overhead with its fixed-length header fields. Also, reducing the complexity of *DTLS* by simplifying the protocol, providing suitable ciphersuites, or supporting *raw public keys* will be a crucial optimisation for *DTLS* to become a valuable technique for securing a constrained environment in the future.

As a first step towards a secured *CoAP*, we implemented *DTLS* for *Californium (Cf)*, a *CoAP* framework written in Java and developed for the use in unconstrained environments. The basic functionalities of the implementation have been successfully tested for interoperability with three other *DTLS* implementations, while *fragmentation* and *raw public keys* remain untested due to current lack of other implementations supporting these. A full handshake increases the average round-trip time of about 40ms by 130 up to 230ms, depending on the chosen ciphersuite. Once a session has been established between two nodes, the overhead to provide security is just 5ms. Based on threat modelling, this implementation secures a connection between two nodes during the operational phase by providing confidentiality and message integrity, and therefore prevents *eavesdropping*, *man-in-the-middle*, and *denial-of-service* attacks. Protection against physical attacks, such as *node capture*, and security bootstrapping remain unresolved for this approach, but potential procedures have been described to achieve both.

Based on the gained experience during the design, implementation, and testing phase, we gave a summary about the potential pitfalls when implementing *DTLS*, and estimated the code and memory complexity of future *DTLS* implementations in low-level programming languages.

Bibliography

[1] ANSI X9.62-2005. *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. American National Standards Institute, Nov. 2005.

[2] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Introducing TakaTuka – A Java Virtual Machine for Motes. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys 2008)*, New York, USA, Nov. 2008.

[3] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, Aug. 2010.

[4] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard), Mar. 2005. Updated by RFCs 4581, 4982.

[5] A. Becher, Z. Benenson, and M. Dornseif. *Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks*, volume 3934 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin / Heidelberg, 2006.

[6] 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. Updated by RFC 5246.

[7] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. draft-ietf-core-block-09, Aug. 2012.

[8] W. Colitti, K. Steenhaut, N. D. Caro, B. Buta, and V. Dobrota. REST Enabled Wireless Sensor Networks for Seamless Integration with Web Applications. In *Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011)*, Valencia, Spain, Oct. 2011.

[9] 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.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[11] O. Dubuisson and P. Fouquart. *ASN.1: Communication Between Heterogeneous Systems*. Morgan Kaufmann, San Francisco, CA, USA, 2001.

[12] P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), Dec. 2005.

[13] O. Garcia-Morchon, S. Keoh, S. Kumar, R. Hummen, and R. Struik. Security Considerations in the IP-based Internet of Things. draft-garcia-core-security-04, Mar. 2012.

[14] D. Giusto, A. Iera, G. Morabito, L. Atzori, C. M. Medaglia, and A. Serbanati. An Overview of Privacy and Security Issues in the Internet of Things. In *The Internet of Things*, pages 389–395. Springer New York, 2010.

[15] K. Hartke and O. Bergmann. Datagram Transport Layer Security in Constrained Environments. draft-hartke-core-codtls-02, July 2012.

[16] C. Hartung, J. Balsalle, and R. Han. Node Compromise in Sensor Networks: The Need for Secure Systems. Technical Report CU-CS-990-05, Department of Computer Science, University of Colorado, Boulder, Jan. 2005.

[17] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), Aug. 2012.

[18] L. Huai, X. Zou, Z. Liu, and Y. Han. An Energy-Efficient AES-CCM Implementation for IEEE802.15.4 Wireless Sensor Networks. In *Proceedings of the International Conference on Networks Security, Wireless Communications and Trusted Computing (NSWCTC 2009)*, Wuhan, Hubei China, Apr. 2009.

[19] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), Sept. 2011.

[20] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), Dec. 2005.

[21] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011)*, Valencia, Spain, Oct. 2011.

[22] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. Updated by RFC 6151.

[23] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN 2011)*, Chicago, USA, Apr. 2011.

[24] C. Lerche, K. Hartke, and M. Kovatsch. Industry Adoption of the Internet of Things: A Constrained Application Protocol Survey. In *Proceedings of the 7th International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2012)*, Kraków, Poland, Sept. 2012.

[25] D. McGrew and D. Bailey. AES-CCM Cipher Suites for Transport Layer Security (TLS). RFC 6655 (Proposed Standard), July 2012.

[26] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), Sept. 2007. Updated by RFC 6282.

[27] N. Modadugu and E. Rescorla. The Design and Implementation of Datagram TLS. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2004)*, San Diego, USA, Feb. 2004.

[28] S. Park, K. Kim, W. Haddad, S. Chakrabarti, and J. Laganier. IPv6 over Low Power WPAN Security Analysis. draft-daniel-6lowpan-security-analysis-05, Mar. 2011.

[29] D. Pauli and D. I. Obersteg. *Californium*. Lab Project, D-INFK, ETH Zurich, Dec. 2011.

[30] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha. Analyzing the Energy Consumption of Security Protocols. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design (ISLPED 2003)*, Seoul, Korea, Aug. 2003.

[31] S. Raza, S. Duquennoy, J. Höglund, U. Roedig, and T. Voigt. Secure Communication for the Internet of Things - A Comparison of Link-Layer Security and IPsec for 6LoWPAN. *Security and Communication Networks*, Jan. 2012.

[32] S. Raza, D. Trabalza, and T. Voigt. 6LoWPAN Compressed DTLS for CoAP. In *Proceedings of the 8th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2012)*, Hangzhou, China, May 2012.

[33] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012.

[34] A. Samiah, A. Aziz, and N. Ikram. An Efficient Software Implementation of AES-CCM for IEEE 802.11i Wireless Standard. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Beijing, China, July 2007.

[35] B. Sarikaya, Y. Ohba, R. Moskowitz, Z. Cao, and R. Cragie. Framework for Securely Setting Up Smart Objects. draft-sarikaya-solace-setup-framework-00, Sept. 2012.

[36] B. Sarikaya, Y. Ohba, R. Moskowitz, Z. Cao, and R. Cragie. Security Bootstrapping Solution for Resource-Constrained Devices. draft-sarikaya-core-sbootstrapping-05, July 2012.

[37] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). draft-ietf-core-coap-12, Oct. 2012.

[38] Standards for Efficient Cryptography Group (SECG). SEC 2: Recommended Elliptic Curve Domain Parameters, Sept. 2000.

[39] Standards for Efficient Cryptography Group (SECG). SEC 1: Elliptic Curve Cryptography, May 2009.

[40] The Internet Assigned Numbers Authority (IANA). Transport Layer Security (TLS) Parameters. Cipher Suite Registry 18 July 2012.

[41] R. H. Weber. Internet of Things – New Security and Privacy Challenges. *Computer Law Security Review*, 26(1):23–30, Jan. 2010.

[42] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610 (Informational), Sept. 2003.

[43] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), Mar. 2012.

[44] P. Wouters, J. Gilmore, S. Weiler, T. Kivinen, and H. Tschofenig. Out-of-Band Public Key Validation for Transport Layer Security. draft-ietf-tls-oob-pubkey-04, July 2012.

A Handshake Protocol

This chapter gives a detailed view about the *DTLS* handshake protocol. Following, a list of concepts used during the handshake that need explanation:

Pseudorandom Function (PRF)
: A *Pseudorandom Function* takes as input a secret, a seed, and an identifying label and produces an output of arbitrary length. It is mainly used to generate keying material.

Premaster Secret
: The *Premaster Secret* is a shared secret between two communicating peers. Its derivation depends upon the negotiated key exchange algorithm: It can be predefined and already shared between the two parties or it can be generated during the handshake protocol (e.g., using *Diffie-Hellman*). It can be of arbitrary length.

Master Secret
: The *Master Secret* is a 48-byte secret shared between the two peers in the connection. It is generated using the *Premaster Secret*, a 32-byte random value provided by each peer and a *PRF*. This step is required because the *Premaster Secret* might not provide enough entropy to derive the keys used for encryption and message authentication.

Connection States
: A *Connection State* specifies a compression, encryption and *MAC* algorithm, and the corresponding keys. There are always four connection states outstanding: The *current* read and write state and the *pending* read and write state. All records are processed under the current read and write states. The handshake protocol is used to set the pending state and the *ChangeCipherSpec* can make either of the pending states current: When a peer sends a *ChangeCipherSpec* it will make its *pending* write state the *current*, meaning that outgoing records are now processed by the new state. Receiving such a *ChangeCipherSpec* makes the *pending* read state the *current* one.

A.1 Full Handshake

Figure A.1 illustrates the most complex type of the *DTLS* handshake protocol containing all possible handshake messages: *ECDH Key Exchange* with mutual authentication. The dotted arrows indicate the messages that can be omitted by protocol definition; if left out,

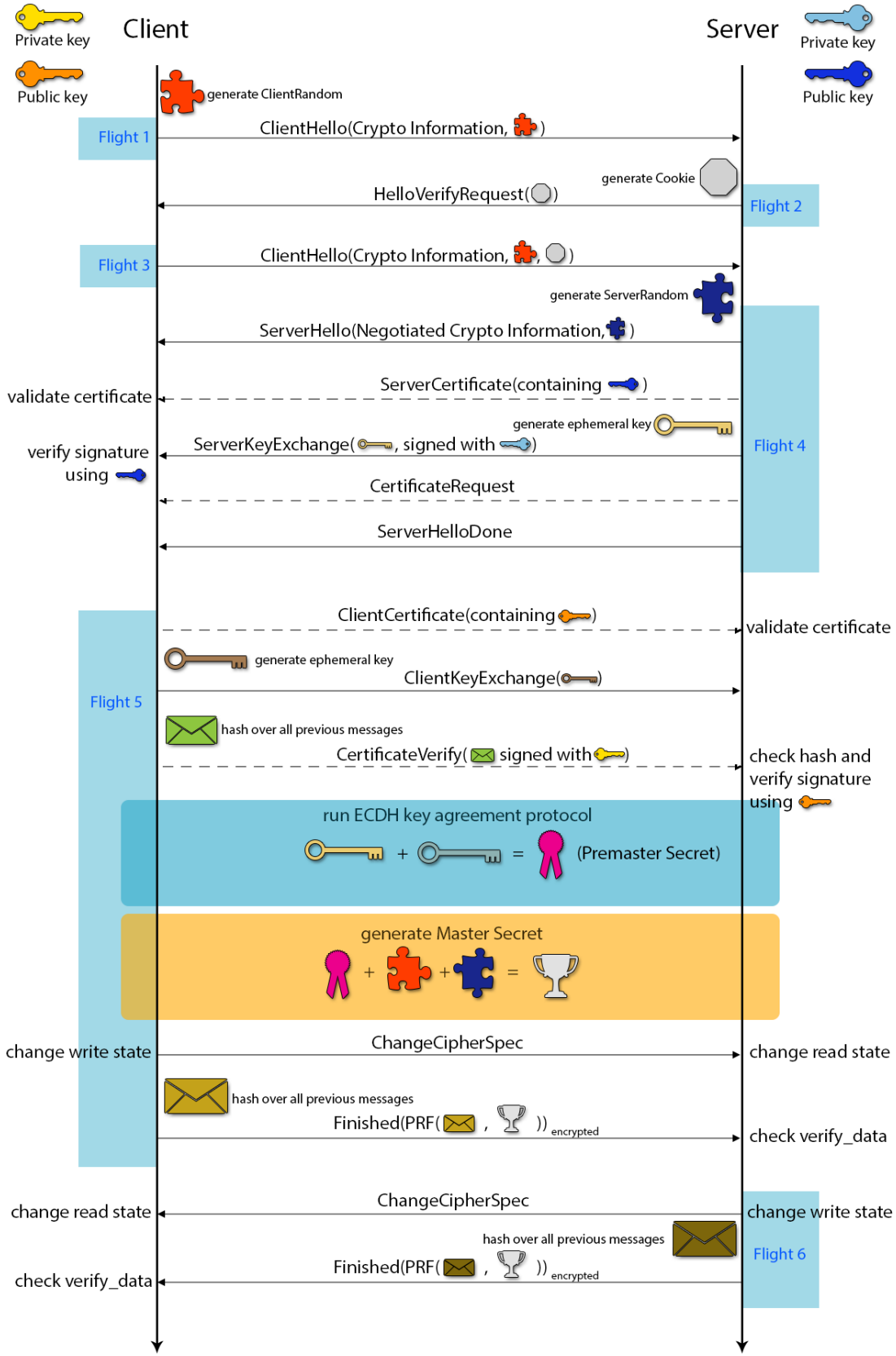


Figure A.1: Message flights for a full DTLS handshake using ECDH key agreement and mutual authentication.

neither the client nor the server will be authenticated at the end of the handshake. The only difference between the *DTLS* and the *TLS* handshake protocol is the message sent at *Flight 2*, the *HelloVerifyRequest* to prevent *DoS* attacks.

Please note that the handshake for *Pre-Shared Key* ciphersuites is a simplified version of the depicted protocol, omitting the mutual authentication (*ServerCertificate*, *CertificateRequest*, *ClientCertificate* and *CertificateVerify*) and the *ServerKeyExchange* is excluded as well. The *Premaster Secret* is derived by the identity of the Pre-Shared Key provided in the *ClientKeyExchange* making the *ServerKeyExchange* irrelevant. The following listing will provide explanations for each handshake message sent in a full *DTLS* handshake with used ciphersuite `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` and mutual authentication.

ClientHello 1

The *ClientHello* message initiates a full handshake with a server. The client provides therein the security parameters it wants to run the handshake upon (i.e., *DTLS* version, preferred ciphersuites, compression methods and several extensions). Additionally, it provides a 32-byte random value which will be used in a later step of the protocol to compute the *Premaster Secret*. In the first version of this message, the *Cookie* field will be empty.

HelloVerifyRequest

The *HelloVerifyRequest* has been added by the *DTLS* specification as a *DoS* countermeasure [33]. There are particular two attacks that can be prevented with this technique: a) An attacker can cause the server to allocate state and potentially perform expensive cryptographic operations by transmitting a series of handshake initiation requests. b) An attacker can use the server as an amplifier by impersonating the victim, resulting in the server flooding the victim with possible quite large messages (e.g., *ServerCertificate*).

The *HelloVerifyRequest* contains a stateless cookie which is sent back to the client, forcing the client to be able to receive the cookie which makes *DoS* attacks with forged addresses hard.

ClientHello 2

This *ClientHello* is identical to the first one sent, except for the added cookie taken from the *HelloVerifyRequest*.

ServerHello

The *ServerHello* message is used by the server to notify the client about the negotiated security parameters under which the handshake will be executed (i.e., *DTLS* version, ciphersuite and compression method). Additionally, the server provides a 32-byte random value, also used later for calculating the *Premaster Secret*.

ServerCertificate

This message contains the server's certificate chain ultimately signed by a certificate authority. The first certificate in this chain contains the server's static *ECDH*-capable public key and is signed with the *ECDSA* [6]. Upon receiving this message, the

client validates the certificate chain and extracts the server's public key which will be used to verify the *ServerKeyExchange* message.

ServerKeyExchange

The *ServerKeyExchange* message is used to convey the server's ephemeral *ECDH* public key to the client and the used elliptic curve domain parameters. The message is signed with the private key corresponding to the public key sent in the *ServerCertificate* message. Upon receipt, the client verifies the signature and extracts the server's ephemeral public key and the elliptic curve domain parameters.

For a description about the *Elliptic Curve Cryptography (ECC)* please refer to Appendix B.

CertificateRequest

This message is sent whenever a non-anonymous server requires the client to authenticate itself as well. It specifies a list of certificate types that the client may offer, a list of hash/signature algorithms the server is able to verify and list of acceptable *CAs*. Only when this message is sent by the server, mutual authentication can be achieved.

ServerHelloDone

The *ServerHelloDone* message indicates that the server is done sending messages to support the key exchange. It is needed because otherwise the client cannot be sure if the server's flight is finished, because the second to last message defined in the protocol (*CertificateRequest*) is not mandatory to be sent by the server.

ClientCertificate

If required by the server (by sending the *CertificateRequest* message), the client needs to send its certificate chain to the server to convey its static public key. The server validates the certificate chain and extracts the client's public key.

ClientKeyExchange

This message is sent in all key exchange algorithms. In this mode (`ECDHE_ECDSA`) it contains the client's ephemeral *ECDH* public key which it created corresponding to the parameter it received from the server in the *ServerKeyExchange* message. Upon receipt, the server extracts the client's ephemeral key. Now, after having exchanged their public key of the *ECDH* key pair, the client and the server can now run the *ECDH* key agreement protocol (described in Appendix B) to obtain the *Premaster secret*.

CertificateVerify

This message is only sent following a *ClientCertificate* that has signing capability. The meaning for this message is to prove that the client really possesses the private key corresponding to the public key in the *ClientCertificate* message. The client computes the hash over all handshake messages sent or received so far and computes

the signature of it using its private key corresponding to the public key in the *ClientCertificate*. The server verifies the signature using the client's public key received in the *ClientCertificate* message.

ChangeCipherSpec (Client)
After having computed the *Master Secret* the client signals that each message received after this one will be secured with the negotiated security parameters.

Finished (Client)
The *Finished* message is the first one protected with the newly negotiated security parameters. The client hashes all previous sent or received handshake messages (excluding the first *ClientHello*, *HelloVerifyRequest* and *ChangeCipherSpec*) and generates the `verify_data` by applying it to the *Pseudorandom Function (PRF)*. The server validates the client's `verify_data`.

ChangeCipherSpec (Server)
After having the client's *Finished* message validated, the server sends its *ChangeCipherSpec* with the same semantics as the client's. Therefore, the pending write state is copied to the current write state.

Finished (Server)
The server's *Finished* message contains the `verify_data` which is computed the same way as before, but the hash of the handshake messages also contains the client's *Finished* message. This is the server's first message protected with the negotiated security parameters, as the message directly follows its *ChangeCipherSpec* message.

Once a side has sent its *Finished* message and received and validated the peer's *Finished* message, it may begin to send and receive application data over the connection.

For further details please refer to the specifications of *ECC* [6], *TLS* [10] and *DTLS* [33].

A.2 Abbreviated Handshake

An abbreviated handshake can be executed when the client and server decide to resume a previous session. The message flow for this scenario is illustrated in Figure A.2. It can only be executed when the two peers have successfully finished a full handshake in the past, thus, having stored a session containing the session identifier and the *Master Secret* generated in the full handshake.

The Client sends a *ClientHello* using the session identifier of the session to be resumed. The server checks its session cache for a match. If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a ServerHello with the same session identifier value. The fresh keys are generated from the existing *Master Secret* and the newly exchanged random values. After the *ChangeCipherSpec* messages, the connection will be secured with the fresh keys and the security parameters negotiated during the full handshake. If the server does not find a match in the session cache, it generates a new session identifier and initiates a full handshake.

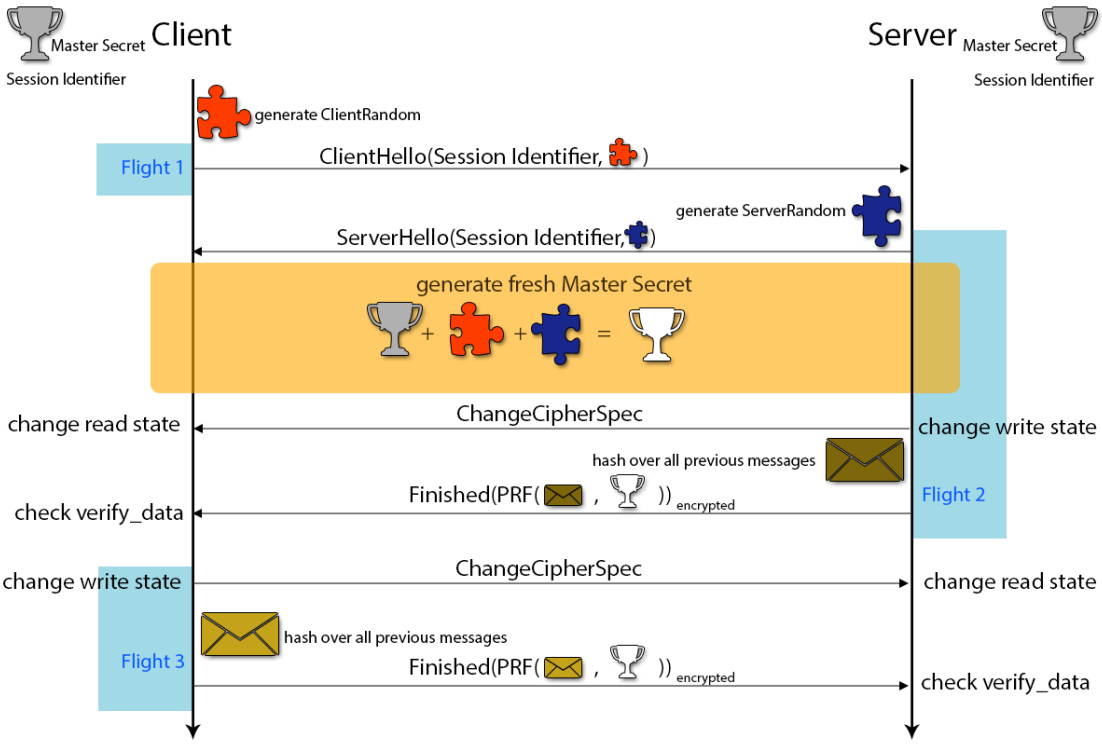


Figure A.2: Message flights for an abbreviated DTLS handshake.

B Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) represents an alternative approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields specified by the *Standards for Efficient Cryptography Group (SECG)* [39]. *ECC* involves three schemes:

- Signature schemes,
- encryption and key transport schemes, and
- key agreement schemes.

Since the implemented ciphersuites only make use of *ECC* as a key exchange algorithm, we elaborate on the signature schemes and key agreement schemes only.

B.1 ECC-Based Key Exchange Algorithms

There are several key exchange algorithms proposed in RFC 4492 to compute the *DTLS Premaster Secret*. The one we are interested in is called `ECDHE_ECDSA`: ephemeral *ECDH* with *ECDSA* signatures [6]. The server and client exchange their ephemeral *ECDH* public keys which both have been generated on the same negotiated elliptic curve. The server's parameters must be signed with *ECDSA* using the private key corresponding to the public key in the server's certificate (see Appendix A). Both client and server perform the key agreement operation and use the result as the *premaster secret*.

B.2 Key Agreement Scheme

The *Elliptic Curve Diffie-Hellman* scheme is a key agreement scheme based on *ECC* which allows two entities to establish a shared secret over an insecure channel [39]. Both need an elliptic curve public-private key pair over a specified elliptic curve. This shared secret can be used to derive the *Master Secret* which can then be used for a symmetric key cipher. The algorithm consists of three steps:

Scheme Setup

Both parties need to establish the elliptic curve domain parameters $T = (p, a, b, G, n, h)$ [39]. This is done in our implementation using the *ServerKeyExchange* message which contains the `named_curve` field: One value out of a predefined set of elliptic curve domain parameters.

Key Deployment

The client and the server both generate an elliptic curve key pair (d_c, Q_c), and (d_s, Q_s) respectively, with the negotiated elliptic curve domain parameters T. They now need the exchange their private keys Q_c and Q_s. This is done in the handshake protocol using the *ClientKeyExchange* and the *ServerKeyExchange* message.

Key Agreement Operation

To derive the shared secret, the client computes $(x_k, y_k) = d_c Q_s$ and the server $(x_k, y_k) = d_s Q_c$. The shared secret is now x_k, the x-coordinate of the computed point.

B.3 Signature Scheme

ECDSA is a variation of the *Digital Signature Algorithm (DSA)* that uses *ECC*. The data to be signed or verified is hashed and the result is run directly through the *ECDSA* algorithm with no additional hashing. *SHA-1* is used as the default hash function with a digest size of 160 bits. The *ECDSA* algorithm is performed according to ANSI X9.62 [1].

B.4 Advantages

There is a certain reason why the *CoAP* draft requires one of the two mandatory ciphersuites to use *ECC*: Compared to the commonly used cryptosystems (e.g., *RSA*), *ECC* can use smaller key sizes to achieve the same level of security. Table B.1 [6] shows the significant smaller key sizes that can be used by *ECC* which results in savings for power, memory (i.e., storing of the keys), bandwidth (i.e., smaller *ServerKeyExchange* and *ClientKeyExchange* messages) and computational cost that make *ECC* attractive for constrained environments [6].

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Table B.1: Comparable key sizes (in bits) for different cryptosystems.

C Keytool and OpenSSL

This chapter illustrates how to generate a certificate chain signed by a trusted authority using the `keytool` and `openssl` commands. We will create a certificate chain as depicted in Figure C.1: There will be a certificate with the alias *end* that contains the entity's public key to which a corresponding private key exists. This certificate will be signed by an intermediate *CA* named *inter* which is signed by a trusted *CA* with the alias *root* that completes the chain of trust. The *end* and *inter* certificate can then be sent to other parties (which trust the *root CA* as well) to authenticate itself.

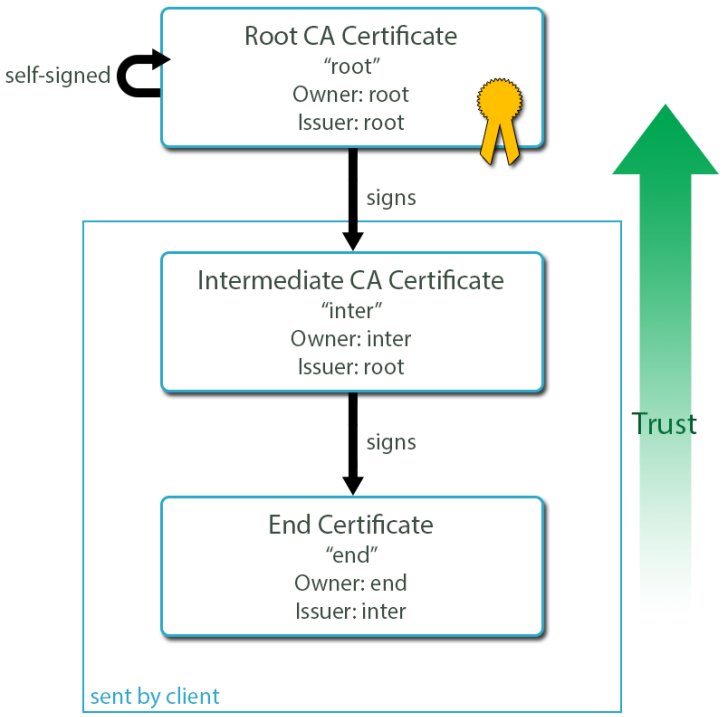


Figure C.1: Example for a signed certificate chain. On top the trusted root certificate authority.

Generate the Private Key of the CA

We generate private keys for both *root* and *inter* in `openssl` since `keytool` is not capable of signing *Certificate Signing Requests (CSRs)*.

```
openssl ecparam -name secp521r1 -genkey -out root.key
```

```
openssl ecparam -name secp521r1 -genkey -out inter.key
```

Generate Self-Signed Root Certificate

The trusted root *CA* signs its certificate with its own private key (`root.key`).

```
openssl req -new -key root.key -x509 -days 365 -out root.crt
```

This gives us a self-signed X.509 certificate (`root.crt`) which is valid for 365 days.

Generate CSR for Intermediate CA

```
openssl req -new -key inter.key -out inter.csr
```

This creates a *CSR* (`inter.csr`) which can now be issued to the root *CA* to be signed.

Sign the CSR with the Private Key of the Root CA

```
openssl x509 -req -in inter.csr -CA root.crt -CAkey root.key -out
    inter.crt -days 365 -CAcreateserial
```

The Intermediate *CA* has now a certificate which is signed by the root *CA*. Naturally, in a real-world setup, this *CSR* would be signed by a commercial *CA* (such as *Thawte* or *VeriSign*) that is trusted already by many devices.

Generate Key Pair for End Certificate

```
keytool -genkeypair -alias end -keyalg EC -keystore keyStore.jks
   -sigalg SHA1withECDSA -validity 365
```

This generates an *ECC* key Pair which is stored in the `keyStore.jks` file.

Generate CSR for End Certificate

We create now a *CSR* which contains the public key which can then be signed by the intermediate *CA*.

```
keytool -certreq -alias end -keystore keyStore.jks -file end.csr
```

Sign the CSR with the Private Key of the Intermediate CA

```
openssl x509 -req -in end.csr -CA inter.crt -CAkey inter.key -out
    end.crt -days 365 -CAcreateserial
```

We now have an end certificate (`end.crt`) which is signed by the intermediate *CA*.

Add the Root Certificate to the Trusted Certificates

Since the Root certificate is a self-signed certificate, there is no chain of trust which can be invoked, because every entity can sign its own certificate. Therefore, we need to add the certificate to the trusted certificates. For this use, there exists a keystore file named *cacerts*, which resides in the security properties directory of Java. It comes with several root *CA* certificates already installed (such as *Thawte* or *VeriSign*) and can be configured by the system administrator.

```
keytool -importcert -alias californium -file root.crt -keystore
   java.home/lib/security/cacerts
```

Now, we need other parties to trust our root certificate as well, otherwise they will not be able to validate our certificate chain because of the missing trust in the root. Therefore, we add our root certificate to a truststore which must then be delivered securely to remote parties. By importing our root certificates to their trusted certificates, the chain of trust can be established during the handshake protocol.

```
keytool -importcert -alias root -file root.crt -keystore
   trustStore.jks
```

Import the Certificate Chain into the Keystore

First, we need to import the intermediate certificate into the keystore because the chain of trust relies on this certificate such that the end certificate can be imported (and trusted) afterwards.

```
keytool -importcert -alias inter -file inter.crt -keystore
   keyStore.jks -trustcacerts
```

The `-trustcacerts` parameter specifies that additional certificates are considered for the chain of trust, namely the certificates in the file *cacerts*, in which we added the root certificate.

```
keytool -importcert -alias end -file end.crt -keystore keyStore.
   jks
```

List the Certificates

```
keytool -list -v -keystore keyStore.jks
```

This command shows us the details of the created keystore, see Listing C.1 for the output. The keystore contains a private key and a certificate containing the corresponding public key. Additionally, there is a certificate chain with a trusted root certificate that signs the intermediate one which signs the end certificate.

Listing C.1: Content of the keystore file.

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 2 entries

Alias name: inter
Creation date: 11.09.2012
Entry type: trustedCertEntry

Owner: CN=Californium Intermediate, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Issuer: CN=Californium Root, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Serial number: a4a217c669b9b966
Valid from: Tue Sep 11 11:58:21 CEST 2012 until: Wed Sep 11 11:58:21 CEST 2013
Certificate fingerprints:
         MD5:  42:24:83:50:45:EE:B2:C7:12:4F:23:35:87:21:F2:0F
         SHA1: FB:F2:83:61:E1:EE:63:F9:B6:D3:BD:90:66:12:37:4F:15:8E:3D:3B
         SHA256: 6A:9F:AE:87:D2:4C:EB:AC:D1:24:DB:96:84:DE:C0:2B:94:74:56:53:5D:72:4A
             :14:FD:CC:D2:0D:2E:C5:F6:78
         Signature algorithm name: SHA1withECDSA
         Version: 1

*******************************************
*******************************************

Alias name: end
Creation date: 11.09.2012
Entry type: PrivateKeyEntry
Certificate chain length: 3
Certificate[1]:
Owner: CN=Californium End, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Issuer: CN=Californium Intermediate, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Serial number: be24ae1b075debaa
Valid from: Tue Sep 11 11:59:42 CEST 2012 until: Wed Sep 11 11:59:42 CEST 2013
Certificate fingerprints:
         MD5:  D5:01:26:E4:8D:2E:51:2D:16:76:64:7F:82:79:78:E0
         SHA1: 44:35:E7:21:43:81:E5:E2:7D:5C:B4:B2:A6:53:7B:0E:85:F8:20:83
         SHA256: 07:13:52:5D:D7:AC:A0:05:F5:89:53:96:8C:A0:5A:AE:8F:18:DC:76:2B:2C:C4
             :09:F1:1C:EB:10:28:A7:DE:68
         Signature algorithm name: SHA1withECDSA
         Version: 1
Certificate[2]:
Owner: CN=Californium Intermediate, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Issuer: CN=Californium Root, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Serial number: a4a217c669b9b966
Valid from: Tue Sep 11 11:58:21 CEST 2012 until: Wed Sep 11 11:58:21 CEST 2013
Certificate fingerprints:
         MD5:  42:24:83:50:45:EE:B2:C7:12:4F:23:35:87:21:F2:0F
         SHA1: FB:F2:83:61:E1:EE:63:F9:B6:D3:BD:90:66:12:37:4F:15:8E:3D:3B
         SHA256: 6A:9F:AE:87:D2:4C:EB:AC:D1:24:DB:96:84:DE:C0:2B:94:74:56:53:5D:72:4A
             :14:FD:CC:D2:0D:2E:C5:F6:78
```

```
         Signature algorithm name: SHA1withECDSA
         Version: 1
Certificate[3]:
Owner: CN=Californium Root, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Issuer: CN=Californium Root, OU=ETHZ, O=ETHZ, L=Zurich, ST=Zurich, C=CH
Serial number: 88cca4960325f71c
Valid from: Tue Sep 11 11:57:28 CEST 2012 until: Wed Sep 11 11:57:28 CEST 2013
Certificate fingerprints:
         MD5:  ED:C9:B6:D6:B9:78:07:D3:07:ED:5E:75:B6:2A:53:C5
         SHA1: 7E:25:0B:35:FB:72:CA:70:AD:39:20:41:7D:71:48:23:38:74:C2:73
         SHA256: 21:F8:56:82:2C:74:5D:BB:19:FC:93:70:58:0B:85:FC:5C:11:73:DE:C1:D2:B9
             :96:70:25:BB:60:85:AB:50:66
         Signature algorithm name: SHA1withECDSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 1B 0A 85 DE A1 9A 30 D8   42 6B 7A 09 A4 6A 03 A7  ......0.Bkz..j..
0010: FB 91 D9 94                                        ....
]
]

#2: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 1B 0A 85 DE A1 9A 30 D8   42 6B 7A 09 A4 6A 03 A7  ......0.Bkz..j..
0010: FB 91 D9 94                                        ....
]
]

#3: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

*****************************************
*****************************************
```