

Master's Thesis

# Compact Object Security for the Internet of Things

Joakim Brorsson  
Martin Gunnarsson



Department of Electrical and Information Technology,  
Faculty of Engineering, LTH, Lund University, 2016.

# Compact Object Security for the Internet of Things

Joakim Brorsson  
ada10jbr@student.lu.se  
Martin Gunnarsson  
dat11mgu@student.lu.se

Department of Electrical and Information Technology  
Lund University

Advisors:  
Ludwig Seitz  
Martin Hell

June 29, 2016

Printed in Sweden  
E-huset, Lund, 2016

---

# Abstract

---

The Internet of Things is coming. With it comes security challenges not present on common, more capable, devices such as desktop computers or servers. We argue that traditional channel security needs to be complemented with object security to cope with the constrained nature of small devices and Low Power Lossy Networks. The main reason for a partial transition to object security being heavy use of asynchronous communication. This thesis explores the feasibility of OSCoAP, a novel draft for an object security solution for CoAP, on constrained devices. It also evaluates the performance of OSCoAP compared to the well known channel security standard DTLS. We find that OSCoAP is indeed implementable on constrained devices and that it actually outperforms DTLS in some aspects. Further, we suggest some minor alterations to the proposed draft.

## Keywords

Internet of Things, Object Security, Constrained Devices, CoAP, OSCoAP.



---

# Acknowledgements

---

We would like to express our deepest thanks and gratitude to our supervisor Ludwig Seitz at SICS, for guiding us through this project. He has provided invaluable guidance and unfailing enthusiasm through the entire process. We would also like to thank our supervisor Martin Hell at EIT for valuable input. Further thanks goes to the rest of the employees at SICS and to Francesca Palombini and Göran Selander at Ericsson for their help with technical matters and for giving us their insights and feedback.



---

# Table of Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Objectives	1
1.2	Related work	3
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Terminology	5
2.2	Characteristics of IoT networks	6
2.3	Problem statement	7
<b>3</b>	<b>Current technology</b>	<b>11</b>
3.1	CoAP	12
3.2	Channel Security Protocols	13
3.3	IPv6, 6LoWPAN and IEEE 802.15.4	14
<b>4</b>	<b>Protocol description</b>	<b>17</b>
4.1	CBOR	17
4.2	COSE	18
4.3	The AEAD AES-CCM	18
4.4	OSCoAP	19
<b>5</b>	<b>Protocol implementation</b>	<b>25</b>
5.1	Californium, a Java implementation	27
5.2	Erbium CoAP on Contiki OS	29
5.3	Deviations from the OSCoAP draft	30
<b>6</b>	<b>Quantitative methodology</b>	<b>31</b>
6.1	Earlier tests	31
6.2	Testing environment	32
6.3	Methodology	32
<b>7</b>	<b>Results</b>	<b>35</b>
7.1	Network overhead	35
7.2	CPU time	37
7.3	Memory footprint	38



<b>8 Discussion</b>	<b>41</b>
8.1 Network overhead . . . . .	41
8.2 CPU-Time . . . . .	42
8.3 Memory footprint . . . . .	42
8.4 Deviations from OSCoAP draft 4 . . . . .	43
8.5 Similar approaches . . . . .	44
8.6 Future Work . . . . .	44
<b>9 Conclusion</b>	<b>47</b>
<b>References</b>	<b>49</b>

---

## List of Figures

---

2.1	End-to-end security compared to Hop-by-hop security. . . . .	8
3.1	Constrained REST stack (CoAP). . . . .	11
3.2	Non constrained REST stack (HTTP). . . . .	11
3.3	CoAP message format. . . . .	12
3.4	DTLS Handshake . . . . .	15
4.1	COSE Object . . . . .	19
4.2	OSCoAP and COSE relation . . . . .	22
4.3	CoAP token handling . . . . .	23
5.1	Logical flow for serializing an OSCoAP message. . . . .	25
5.2	Logical flow for parsing an OSCoAP message. . . . .	26
5.3	Californium processing stack. . . . .	28
7.1	One request, one response scenario . . . . .	35
7.2	Response network overhead . . . . .	37



---

## List of Tables

---

4.1	Table showing how OSCoAP should protect options . . . . .	21
7.1	DTLS Handshake sizes. . . . .	36
7.2	Sizes for GET requests and responses with 5 byte payload. . . . .	36
7.3	Execution time of selected functions. . . . .	38
7.4	Execution time for memory functions . . . . .	38
7.5	Memory footprint comparison . . . . .	39
7.6	OSCoAP minimum heap memory allocated at runtime . . . . .	39
7.7	Code size for different parts of OSCoAP . . . . .	40



---

# Introduction

---

The Internet of Things is a term used to describe the increasing number of connected embedded systems used in a wide variety of applications. These systems are partly or fully composed out of devices, often called nodes, that have very restricted resources. For example they often operate on battery power and have very little memory. Therefore, they are often referred to as *constrained nodes*. These constraints affect the operation of the nodes in significant ways.

Network operations are hard on battery consumption. To cope with this, network operations are bulked up in scheduled sending/receiving time slots. In between the slots, the network is offline. I.e. battery constraints result in offline nodes as the default state. This forces constrained nodes to communicate asynchronously to cope with their resource constraints. The Constrained Application Protocol, CoAP [1], is an application level protocol designed to operate under these circumstances. It is resource efficient and handles the *asynchronous communication* situation by working closely with caching or mirroring proxies. To secure this communication, DTLS [2] has gained traction since it operates over UDP, a transport protocol suitable for asynchronous communication and used by most CoAP implementations.

This brings us to the problem. CoAP is a heavy user of proxying functionality. DTLS encrypts everything in its payload, including CoAP. The use of CoAP will be severely crippled by securing it using DTLS, or any other *channel security* protocol. The situation can be improved by transitioning from channel security to *object security*, where the encryption can be limited to include only data, not proxying information and other functional headers. OSCoAP [3] is a protocol draft aimed at using object security to protect CoAP. It fully encrypts the payload of the message while integrity protecting other headers and leaving proxying information be so that CoAP can function unhindered while still being secure.

## 1.1 Objectives

This thesis aims to explore the possibility of using object security, as opposed to channel security, in constrained environments. In this thesis we test the feasibility and efficiency of a communication security solution proposed by Ericsson<sup>1</sup> and

---

<sup>1</sup><http://www.ericsson.se>

SICS<sup>2</sup>. This solution, called Object Security for CoAP (OSCoAP), provides communication security properties for constrained devices operating in the Internet of Things.

The outcome of the project will be a fully functional reference implementation of the proposed OSCoAP protocol providing end-to-end security in constrained environments. This implementation will be coupled with tests comparing it to the current state of the art solutions such as DTLS/CoAP and plain CoAP in a quantitative study which aims to measure the performance difference in terms of:

- Network overhead and latency
- Memory footprint
- CPU usage

The results can be used to evaluate the suitability of OSCoAP as a complement to DTLS in constrained environments.

This will require us to make use of established standards, such as CoAP, and build security features on top of them. The concrete outputs of the project will be:

- An OSCoAP library written in Java for a desktop OS
- An OSCoAP library written in C for constrained devices running Contiki OS [4]
- A quantitative study as described above

The required properties of the libraries is as stated in the OSCoAP draft [3]

- End-to-end encryption of payload and selected headers
- Integrity protection of payload and selected headers
- Replay protection
- Secure mapping of request to responses

### 1.1.1 Research questions

There are situations where object security is a necessity. Our work aims to answer the questions of whether object security, in the form of OSCoAP, is implementable in practice and how it compares in terms of efficiency in scenarios where security can be obtained using either channel security or object security.

### 1.1.2 Delimitations

The proposed solution will not handle any key infrastructure. It will also not implement the OSCON version of OSCoAP [3] and it will not implement the optional sliding window for sequence numbers. The tests will focus on the most simple scenario of “one-request-one-response”, since this is the main use-case for OSCoAP. Requests with multiple responses will not be tested.

---

<sup>2</sup><http://www.sics.se>

## 1.2 Related work

There are numerous approaches for adapting channel security for constrained devices. Raza et al. have adapted IPsec and DTLS for constrained devices using header compression extensions for 6LoWPAN [5][6]. Hummen et al. suggests offloading parts off the DTLS handshake to trusted gateways [7]. An approach similar to Hummens, but with end to end data integrity, is provided by Sethi et al [8].

The approach to security most similar to ours is OSCAR by Vucinic et al. This solution is, like OSCoAP, based on object security [9]. A discussion on how these related approaches relates to OSCoAP is provided in Section 2.3.2.

### 1.2.1 Prior work on OSCoAP

Palombini, now co-author of the OSCoAP draft, has written a master thesis in 2015 which implemented the message integrity part of OSCoAP draft 2 on Contiki OS [4][10][11]. Our work adds to this work and aims to extend it by implementing draft 4 of OSCoAP which differs significantly from draft 2. This thesis will also evaluate the performance of message confidentiality on top of message integrity. The most significant difference between this and prior work is that since draft 4 of OSCoAP, the message protection is based on COSE. We will clarify the similar and non similar parts in the technical description.





## 2.1 Terminology

### 2.1.1 Channel Security

Channel security is a term used in communications security which describes a secure channel used by an application to transmit data [12]. The channel is negotiated and managed by a protocol at the data, network or transport level in the protocol stack. The channel handles the data agnostically; it does not know anything about the payload.

### 2.1.2 Object Security

Object security is a term used in communications security describing secure communication with no need for a secure channel [12]. Instead of relying on a communication protocol lower in the stack to handle the encryption, the application that created the message will handle encryption and decryption of its own communication.

The difference between channel security and object security may seem subtle, the key difference is that in object security an application handles its own secure communication. This has the effect that when using object security, an application does not need to rely on a secure channel. Rather, it can pick and choose by itself what to protect and how to protect it. This is a very important aspect, as we shall see.

### 2.1.3 Internet of Things and Constrained Nodes

Internet of Things (IoT) is the term used to describe the increasing number of connected embedded systems used in a wide variety of applications. M2M (Machine to Machine) and constrained nodes are also common terms. The distinction between the expressions is somewhat fuzzy but in broad terms both IoT and M2M are networked systems which are comprised partly or fully out of constrained nodes.

A constrained node is a constrained device in a network. A constrained device is defined by RFC7228 [13] as “A node where some of the characteristics that are

otherwise pretty much taken for granted for Internet nodes at the time of writing are not attainable, often due to cost constraints and/or physical constraints on characteristics such as size, weight, and available power and energy.”. The same document gives some concrete numbers for what can be considered a constrained device, for example the RAM memory of a constrained device can range from under 10 KiB to roughly 50 KiB. Common poster children for constrained devices are the Arduino and ARM Cortex family of devices.

M2M networks are networks that consists out of interconnected nodes. The node can be constrained, but that is not always the case. These networks are often thought of as isolated systems with a well defined purpose, for example industrial control systems.

IoT is the term for the connection between different constrained systems. The Internet of Things can be thought of as constrained devices and M2M networks connected to the internet, using it to communicate with each other.

#### 2.1.4 Low-Power Lossy Networks

The network used in a system can also be constrained. If the network is highly unreliable, has a low throughput or another hindering property, the network can be considered constrained. These kind of networks are often referred to as Low-power Lossy Networks (LLNs) [14]. Because of LLNs, systems can be considered constrained even though no nodes are constrained.

## 2.2 Characteristics of IoT networks

The characteristics of constrained devices calls for a different set of solutions than the ones commonly used for non-constrained devices. An interesting area to explore is how to provide information security on constrained devices communicating over insecure channels, i.e, how can message integrity, secrecy, replay protection, message freshness guarantee, and sequence ordering be provided for the Internet of Things.

There are some properties of constrained networks which have significant impact on how communication works compared to how it works on traditional networks. Firstly, communicating parties operating on battery power will naturally want to save battery. Therefore the network part of a constrained device will be turned off for as much time as possible. This means that the normal case will be asynchronous communication and because of this, caching nodes are the norm.

Second, when sending data on the network, you want to send as little data as possible. This is of course always a good thing but it is extra important for constrained nodes; power consumption from network hardware is very high compared to cryptographic operations [15]. Another aspect of minimizing transmitted data is that large data packages will require larger buffers on the recipient node, something that might not be available on constrained nodes. Sending large packages also increases the retransmission amount. If a fragment of a large message is dropped by the network, or there is an error somewhere in a packet, all of

the message might need to be retransmitted instead of just a small part. Packet fragmentation can be a considerable source for network overhead.

## 2.3 Problem statement

### 2.3.1 Traditional methods

Secure communication for constrained nodes can certainly be achieved using traditional methods such as channel security. The security in these traditional protocols protects the channel, not the data itself, hence the name. Since many devices operate on battery power it is important to use as little resources as possible, both in terms of power consumption and memory/CPU usage. The consequences of this is very often seen in devices designed so that they sleep for as much time as possible. This is discussed in the OSCAR paper [9] which argues that the core problems with security in IoT is that Application traffic is asynchronous, which makes caching a requirement for a well functioning network. To achieve this, caching proxies are often used.

### 2.3.2 Consequences of using traditional methods

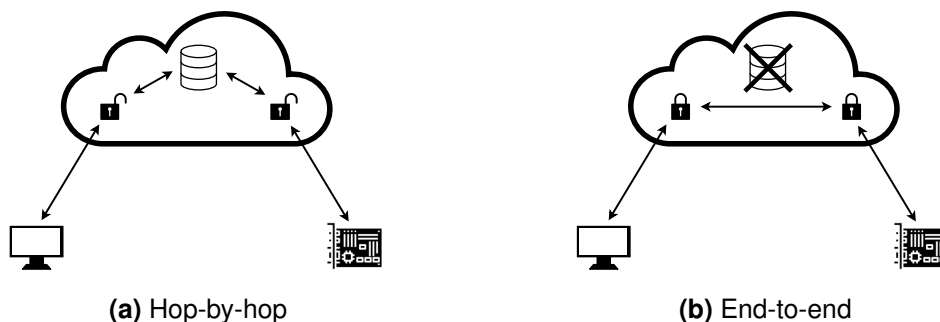
Herein lies the core problem this thesis will discuss. If a proxy is to be used for caching data, and channel security is used, the security can follow two patterns.

The first pattern, hop-by-hop security, visualized in Figure 2.1a, is to terminate the channel security session in the caching node, effectively dividing the security in two parts. One part from the sender to the caching node and one part from the caching node to the receiver. This forces the data to be decrypted at the caching node; the caching node needs to re-encrypt the plaintext for the receiver. The data integrity and confidentiality will therefore not be end-to-end between the client and server, but hop-by-hop from client to proxy and from proxy to server. Hop-by-hop security can only be relied on if all partners are trusted. This is not a good assumption for a secure and robust system. The possibility of malicious nodes opens up for both passive eavesdropping attacks and active attacks such as man-in-the-middle attacks on the communication. These kind of attacks are a very real threat and there has been countless examples of them. Hop-by-hop security is sometimes also referred to as point-to-point security.

The second pattern, end-to-end security, visualised in Figure 2.1b, is to not terminate the session at the proxy but instead keep the channel security enabled through the proxy. This thwarts the possibility for the proxy to attack the session in a meaningful way since it prevents it from reading the data or changing it without detection. True end-to-end security is thereby obtained but important functionality is also lost. With channel security used for end-to-end encryption, it is all or nothing; all data originating from above the session layer has to be secured. The inability for a proxy to change or read anything from the transport layer and higher layers is not without negative consequences. A proxy often carries a lot of functionality on higher layers that is broken by end-to-end channel security. For example, a CoAP caching proxy can not cache any data for connections that

tunnels through the proxy using channel security. CoAP is a protocol designed to work closely with proxies; the protocol will be crippled without the proxying functionality.

It has been suggested to tunnel a DTLS connection through CoAP messages for proxy compatibility [16]. This work has however been claimed to be quite inefficient by the authors themselves, the idea was therefore abandoned.



**Figure 2.1:** End-to-end security compared to Hop-by-hop security

### 2.3.3 Advantages of transitioning to Object Security

When the end-to-end security is based on object security, the protocol can pick and choose what part of the data that should be confidential or integrity protected. For example it would be possible to encrypt the payload, integrity protect static parts of the header and leave variable header parts be. Protecting certain parts of the header can be very important. A malicious intermediate tampering with the header can do considerable damage. For example an attacker could change the *method code* from *GET* to *DELETE*, thereby deleting a resource instead of fetching the current value. This is one reason for why encryption of just the payload of the CoAP message is not sufficient. In conclusion, object security enables end-to-end security without preventing proxy operations, if done correctly.

Another important aspect when comparing object security and channel security for constrained devices is the network overhead produced by the different technologies. The overhead in a session based security protocol is composed of both the handshake and the overhead that the protocol produces when encrypting and integrity protecting data. In an object based security protocol, the encryption parameters can be pre established, in which case the overhead consists solely of the overhead produced when encrypting and integrity protecting, no handshake takes place. We suspect that channel security, DTLS, is more efficient for long running sessions but that object security can be made more efficient for sending individual packets. The DTLS handshake is network intensive. This is true even for the least intensive mode DTLS-PSK, a mode with a Pre Shared Key as a prerequisite, allowing for a more lean handshake [17]. A DTLS handshake will also demand a lot of retransmits on a lossy network. [18] measures the performance of DTLS-PSK on lossy networks and concludes that the success rate of handshakes drops rapidly after a 20 packet loss rate on the network, resulting in retransmits.

An object security based solution will not have this overhead but might not be as efficient in the transmission phase since it will carry a bit more overhead that was not pre-established in a handshake.

### 2.3.4 Proposed solution

A solution more suited to constrained applications is required and SICS in collaboration with Ericsson have therefore proposed using object security in combination with CoAP in their draft for the protocol OSCoAP [3]. In this draft, security of the communication is moved higher up in the protocol stack to provide the needed data confidentiality and integrity. This solution is compliant with the argument of not having to replace existing protocols in unconstrained networks, presented in [19]. However, that paper proposes a solution where end-to-end security is broken in the gateway. Contrary to this, OSCoAP aims at reducing the overhead and complexity by adding object security as an option in CoAP packets.

In this thesis we will argue that the inherent asynchronous properties of constrained devices calls for a transition from connection oriented security solutions to object oriented security solutions in certain scenarios and that OSCoAP is a good way to achieve this.



---

## Current technology

---

A common network stack used for applications on constrained devices is pictured in Figure 3.1. This stack differs significantly from the standard network stack for non constrained devices pictured in Figure 3.2. The constrained stack is the one we use for reference when implementing object security for constrained nodes. The different layers of the constrained stack are explained in more detail below.

CoAP	}	Application layer
DTLS (optional)		
UDP	}	Transport layer
IPv6	}	Network layer
6LoWPAN		
IEEE 802.15.4	}	Physical layer

**Figure 3.1:** Constrained REST stack (CoAP)

HTTP	}	Application layer
TLS (optional)		
TCP	}	Transport layer
IPv4/IPv6	}	Network layer
WiFi/Ethernet	}	Physical layer

**Figure 3.2:** Non constrained REST stack (HTTP)

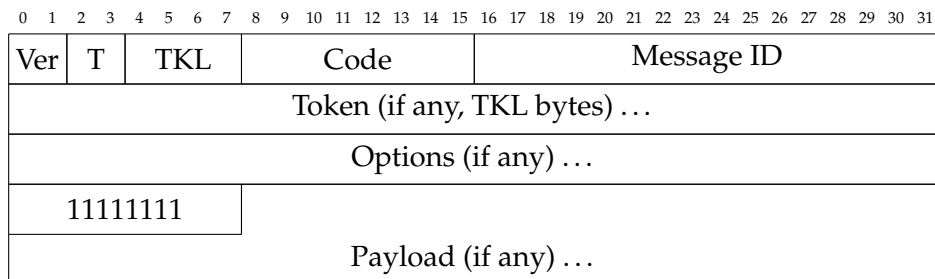


## 3.1 CoAP

The IETF application layer protocol Constrained Application Protocol (CoAP) [1] is a popular REST [20] protocol aimed at network communication in IoT while closely resembling HTTP [21] for easy integration with web services. It features an asynchronous transaction model and has native headers for caching.

### 3.1.1 Message format

The CoAP message is divided into header and payload. The message structure is designed to suit communications where the network or at least one communication partner is constrained. The message structure can be seen in Figure 3.3.



**Figure 3.3:** CoAP message format

The different fields are defined as:

- Version (Ver): CoAP version, currently 01.
- Type (T): Keeps track of whether a message is confirmable, non-confirmable or a confirmation.
- Token Length (TKL): Indicates Length of token field, 0-8 bytes.
- Code: Request/Response code. Analogue to HTTP codes, e.g. GET or POST.
- Message ID: Used for duplication detection and acknowledgement handling.
- Token: Used for Request-Response mapping.
- Options: An ordered list of options related to the message. For example options specifying how a proxy should handle messages and what resource that is requested. Options are explained in more detail below.
- Payload: The payload of the message.

### 3.1.2 Options

The options are a very important part of CoAP. Since the header is short, almost all functions in CoAP are done with options. Options can be used for example to instruct a proxy how to handle the message, set a time frame for how long the message is valid, or even be used for message fragmentation on the application layer. The functionality of CoAP options is very versatile and the CoAP standard is open for adding more options.

The options are transported as part of the CoAP header, where they are stored as a TLV formatted list. TLV is an acronym for Type-Length-Value, which pretty much explains how it functions. Every option present in the list has a unique identifier of fixed length which acts as the Type. The Length field is important because many options can vary in length, URI-Path is one such example. The Length field is used to determine the size of the option data, which resides in the Value field.

### 3.1.3 Proxies

With the asynchronous messaging model used in CoAP, proxies become vital. CoAP is designed to work primarily through proxies. The proxying functionality is implemented through a number of CoAP options. This aspect of CoAP is very important to keep in mind when designing extensions to CoAP or tunnelling CoAP through another protocol.

## 3.2 Channel Security Protocols

### 3.2.1 TLS

TLS is an IETF standard for channel security for the transport layer [22]. It is very common on the web and used in many applications. TLS is most commonly encountered as the S in HTTPS, which is HTTP over TLS. However, it is designed to work over TCP. This makes it unsuitable for constrained devices, which typically use UDP [23].

### 3.2.2 DTLS

DTLS is also an IETF standard for channel based security [1]. It is currently the standard communication security mechanism for CoAP and often encounters as the S in CoAPS. The protocol resembles the TLS standard but is adapted for use with UDP instead of TCP [24]. This is important for constrained nodes since the connectionless transport protocol UDP is more suited for constrained networks.

In rough terms, DTLS functions by first sending a number of packets between 2 communicating nodes to set up a cryptographic key for use in further communications. One node acts as *client* and the other one takes the *server* role. This exchange is performed using the *Handshake Protocol* and the exact behaviour varies depending on preferred cipher suites etc. The handshake procedure is explained

in more detail below. When the handshake phase is over, the communication can commence and be protected using the negotiated key. This protection is carried out using the *Record Protocol*.

### The Record Protocol

The DTLS record protocol is the workhorse caring for security and reliability of message transfers. It can be seen as an encapsulating protocol used to transport data or connection state information. The record layer header contains information about data type, sequence number, length and offset of the content. This information can be used to order and classify incoming content. Other DTLS protocols are carried as payload in a record protocol frame.

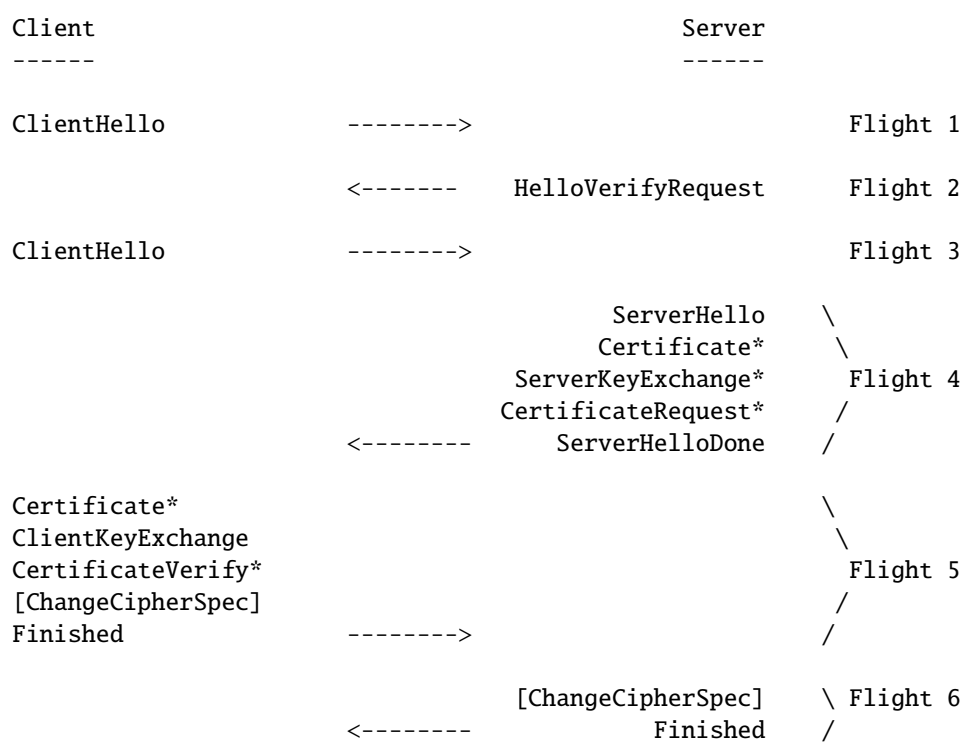
### Handshake Protocol

The handshake protocol is responsible for negotiating a shared key for the communicating parties to use for message protection. Operating on top of the record protocol, a two party procedure ensures a valid connection state used in further communication.

The protocol is initiated by the client sending a *client-hello* message, to which the server replies with a *hello-verify-request*. To prevent replay attacks, DTLS requires a second *client-hello* containing a *nonce* from the *hello-verify-request*. This series of messages are used to assert the security capabilities of the parties. When this is done, the client sends a *client-key-exchange* containing parameters for key negotiation. It also sends a *change-cipher-spec* and a *finished* message, asserting the use of a specific cipher suite. In response to this, the server also sends a *change-cipher-spec* and a *finished* message to acknowledge the start of the protection of messages. With this, the handshake is done. There are more steps to this protocol but these are the most essential ones. The full process is pictured in Figure 3.4.

## 3.3 IPv6, 6LoWPAN and IEEE 802.15.4

IPv6 [25], 6LoWPAN [26] and IEEE 802.15.4 [27] are part of the network stack and support the DTLS and CoAP parts of the stack. They are vital for functionality but their inner workings are not relevant to this thesis.



**Figure 3.4:** Message Flights for Full DTLS Handshake, copied from Figure 1 in RFC6347 [2]. Optional messages are marked with \*.



---

## Protocol description

---

The OSCoAP draft specifies OSCoAP in its entirety. The draft is an individual contribution to the ACE working group at IETF [28]. This means that it is an unfinished document aspiring to become a standard once finished. The specification is dependent on other specifications and drafts described below.

### 4.1 CBOR

Concise Binary Object Representation, CBOR, [29] is a binary, extensible data format with a primary focus on very small code size. Message size is also an important factor in CBOR but secondary to code size. These are very much desired qualities when designing a protocol for constrained devices and therefore CBOR is considered more suitable as a data format for OSCoAP than its competitors such as XML [30] or JSON [31].

The main competitor to CBOR in terms of both code size and message size is MessagePack. There are no implementations for MessagePack for constrained nodes, since it is mainly aimed for usage in a Web environment. We measured the code size of the `cn-cbor` library [32] (constrained node CBOR) and the `cmp` library [33], a small implementation of MessagePack in C. We saw that `cn-cbor` needs about 1.4KB of code for an embedded ARM-Cortex-M3 platform and that `cmp` needs 8.3KB. This is quite a significant difference in favour of CBOR.

JSON is a well used standard which is quite simple. However it is a text based protocol. For the purposes of OSCoAP, a binary protocol is more suited. A combination of JSON and base64 could be used but this would generate network overhead and require more computational power. JSON is also not extensible. There are binary versions of JSON, for example BSON, but these are not satisfactory for various reasons. Some of the more important of these reasons for the strongest opponents are that neither BSON nor MessagePack are official standards [34][35]. Other important reasons are that BSON is more complex and MessagePack has an uncertainty about how extensions will be handled. A comparison of these and other similar protocols can be found in appendix E of the CBOR RFC [29].

XML is also a common standard but it is extremely bloated and, just as JSON, text based. There exists a minimal version of XML called EXI [36] which is quite small in terms of code size and message size, though it carries other disadvantages.

Most importantly, XML/EXI is document oriented while JSON/CBOR is data oriented. This means that to be able to parse XML/EXI you have to be aware of the schema description.

Because of the qualities of CBOR compared to it's competitors, OSCoAP uses CBOR for data representation.

## 4.2 COSE

OSCoAP requires the ability to provide integrity and confidentiality for data. CBOR Object Signing and Encryption, COSE [37] is a well suited data format for this. It is primarily a specification on how to use CBOR to represent content and cryptographic operations on that content, just as the name implies. The data consists of a CBOR object with an array containing entries with the data and cryptographic content.

- The first entry in the array is called the protected field. This field contains information that should be protected by cryptographic operations, for example a partial IV that is to be used for cryptographic operations. This field is always present.
- The second entry in the array is called the unprotected field and contains information that does not need to be protected.
- The third entry is the content of the message. This specifics of the content will consequently vary on what type of data and protection that is used. For example, in a message protected by an AEAD (see section 4.3), this field will contain the ciphertext and the tag.

In Figure 4.1 a COSE-object is shown. This is the COSE object that is sent as part of a OSCoAP response. The key-value pair in the non-empty map is the Sequence Number, the key is 6 and the value is 1. The byte string with length 15 is the ciphertext and tag. The ciphertext is a Content-Format option with value TEXT/PLAIN, the payload delimiter 0xFF and then the payload "hello". Then comes seven bytes of tag.

## 4.3 The AEAD AES-CCM

COSE provides several cryptographic functions to protect messages. OSCoAP only needs to use one of them, namely AES-CCM [38]. AES is the cryptographic primitive used. CCM is a block cipher mode of encryption which does both confidentiality and integrity protection. Using this construct, you can choose which parts that need both confidentiality and integrity protection and which parts should only have integrity protection. This property makes it an algorithm for Authenticated Encryption with Associated Data, an AEAD [39]. With AES-CCM, OSCoAP can use COSE for protecting data in an appropriate way.

```

d9 03e1          # tag(993)
  83            # array(3)
    44          # bytes(4)
      a1        # map(1)
        06      # unsigned(6) - Key[Sequence Number]
          41    # bytes(1)
            01  # bytes(0x01) - Value[1]
      a0        # map(0)
    4f          # bytes(15)
      2cd41d5c67003455351a120e5d1069 # Ciphertext=
                                         [Content-Format=TEXT/PLAIN,
                                         0xFF,
                                         Payload="hello"]

```

**Figure 4.1:** COSE-Object from a OSCoAP response shown in CBOR diagnostic notation

## 4.4 OSCoAP

This section will describe the OSCoAP protocol and the components of the system. OSCoAP is a proposed standard for using COSE to protect CoAP messages.[3] It offers the functionality of message confidentiality and integrity, replay protection and guaranteed message ordering through symmetric key cryptography and sequence numbers. The proposed standard protects both the payload, options and static parts of the CoAP header.

In broad terms, OSCoAP will take the CoAP options along with a potential payload and move them in to a new CoAP option called the object security option. The object security option stores the collected options and payload in a COSE object. Static parts of the header is used to compose Additional Authenticated Data (AAD). The COSE object can then be encrypted and together with the AAD integrity protected using parameters from what is called a security context. A security context contains initialization vectors, sequence numbers and cryptographic keys. Every communication pair, i.e. a server and a client, shares a common security context that is unique for their communication.

### 4.4.1 Sequence numbers

Sequence numbers are included in OSCoAP to provide protection against replay attacks and reordering attacks. A replay attack is a scenario where a malicious user intercepts and saves messages sent between communication partners. The malicious user can then retransmit the saved message at a later time. Without a replay attack defence such as sequence numbers, the message will appear valid. A practical application to this would be that Alice sends a message to her smart door lock to open the door. If Malicious Mallory then intercepts the message, Mallory can later resend the message to open the lock.

Sequence numbers are designed to make this kind of attack impossible. Each communication partner has a pair of numbers, a sending sequence number and a



received sequence number. Every time a message is sent, the sending sequence number is incremented. When a node receives a message it will update its received sequence number if the received number is higher than the current. If the received sequence number is lower than or equal to the saved received number, the message shall immediately be discarded since it is a retransmission of an old message. In the scenario presented above, the door lock would note that the sequence number in Mallory's message is lower than or equal to the saved receiver sequence number. The lock will discard Mallory's message and keep the door locked.

This is the most simple case of handling sequence numbers. OSCoAP can provide a sliding window for sequence numbers and thereby handle messages arriving out of order. The default size of this window is 64 messages. That means that up to 64 messages may be received out of order. Duplicate messages and messages with sequence numbers lower than the lower bound of the sliding window are rejected.

#### 4.4.2 The Security Context

A node communicating with another node using OSCoAP must have a Security Context. The Security Context is a group of predetermined elements needed for cryptographical operations in OSCoAP. The distribution of Security Contexts is out of scope for this work. We assume that every node has a Security Context. The Security Context contains the following:

- Context Identifier, a value that lets the node identify the context.
- Algorithm, The algorithm used to encrypt and authenticate the OSCoAP transaction. AES-128-CCM-8 is the algorithm we use.
- Client Write Sequence Number, used for replay protection.
- Server Write Sequence Number, used for replay protection.
- Client Write Initialization Vector, The IV for the AES-CCM AEAD.
- Server Write Initialization Vector, The IV for the AES-CCM AEAD.
- Client Write Key, The key for the AES-CCM AEAD.
- Server Write Key, The key for the AES-CCM AEAD.
- Replay Window, Specifies how many packets that can arrive out-of-order for the replay protection mechanism.

The security context is present in an exact copy on both client and server. The goal is to keep them as exact copies by incrementing the sequence numbers in the same way on both client and server during a correspondence in order to prevent replay attacks. If a client wants to send an OSCoAP message to a server, the client will protect its message with the "Client Write Key" and the "Client Write Initialization Vector". The server that receives the message will use the same key and IV to decrypt. The server will then send a response protected with the "Server Write Key" and the "Server Write Initialization Vector". The Client will then decrypt the received response with the same keys. The sequence numbers

are used in a similar fashion. The client increments the Client Write Sequence number by one every time it sends a message to the server. The server updates the Client Write Sequence number every time it receives a message from the client. This way the Client Write Sequence number on both ends always represents the most recently sent message. The Server Write Sequence Number works in the same way.

### 4.4.3 Message freshness

OSCoAP guarantees message freshness. To account for proxies not aware of OSCoAP and that such messages shall not be cached, a Max-Age option set to zero can be included. Max-Age is a directive to a caching proxy indicating how long (in seconds) the message can be cached and by setting the value to zero, caching of OSCoAP messages is forbidden.

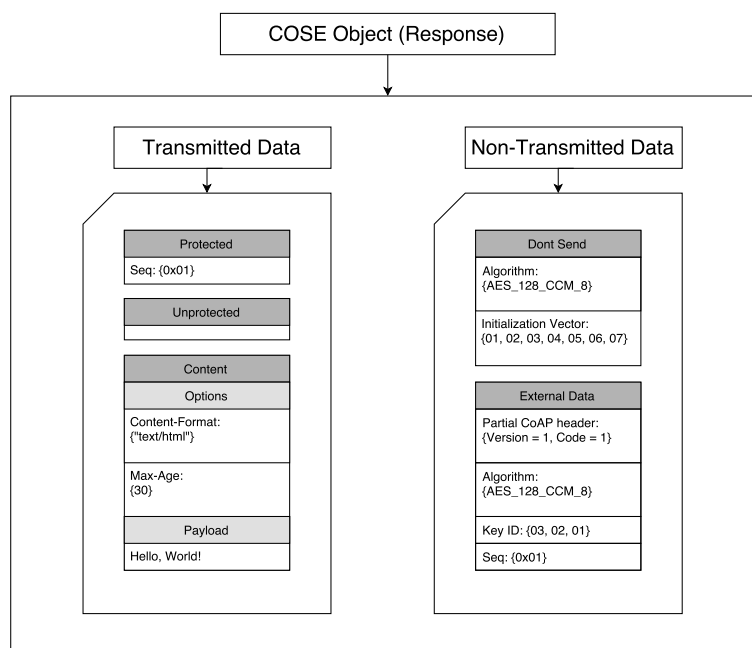
**Table 4.1:** Table showing how OSCoAP should protect options.  
E: encrypt, I: integrity protect, D: can occur both protected and unprotected

No:	C	U	N	R	Name	Format	Length	E	I	D
1	x			x	If-Match	opaque	0-8	x	x	
3	x	x	-		Uri-Host	string	1-255			
4				x	ETag	opaque	1-8	x	x	
5	x				If-None-Match	empty	0	x	x	
6		x	-		Observe	uint	0-2	x	x	x
7	x	x	-		Uri-port	uint	0-2			
8					Location-Path	string	0-255	x	x	
11	x	x	-	x	Uri-Path	string	0-255	x	x	
12					Content-Format	uint	0-2	x	x	
14		x	-		Max-Age	uint	0-4	x	x	x
15	x	x	-	x	Uri-Query	string	0-255	x	x	
17	x				Accept	uint	0-2	x	x	
20				x	Location-Query	string	0-255	x	x	
35	x	x	-		Proxy-Uri	string	1-1034			
39	x	x	-		Proxy-Scheme	string	1-255			
60			x		Size1	uint	0-4	x	x	

### 4.4.4 The Object Security Option and the COSE object

The object security option is a CoAP option that can be seen as a secure data container for the payload and other CoAP options. It also stores data related to the integrity of parts of the header. The object security option uses a COSE-Object to store the data intended to be encrypted and integrity protected.

A visualization of how OSCoAP uses the COSE structure is provided in Figure 4.2. The COSE-Object will hold the options of the CoAP message (specified in Table 4.1), the sequence number of the message and the payload if present.



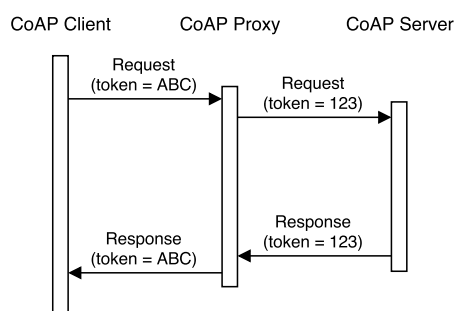
**Figure 4.2:** OSCoAP and COSE relation. All non-transmitted data is either known beforehand, like the algorithm field, or a duplicate of a part in the CoAP header, such as the version or code fields.

If the CoAP message does not have a payload, the COSE-Object will be placed as option-data in the OSCoAP option. If the CoAP message does have a payload, the whole COSE-Object will be sent as the payload of the CoAP message. The COSE object is encrypted and integrity protected with the AES-CCM algorithm before it is sent. The receiving part will decrypt the message and verify the message tag before processing the message further.

#### 4.4.5 Authenticated Data

Not all data in the CoAP message can be confidential, some of it needs to be in plain text. It is however possible to integrity protect that data. An obvious example of what can not be encrypted is the Version field. This field needs to be represented in plain text so that the receiver knows how to process an incoming packet.

The token also needs to be in plain text since it is used for mapping requests to responses. So, just like the Version field, the Token can not be confidential. But contrary to the Version field, the Token can not be integrity protected. This



**Figure 4.3:** CoAP allows a proxy to change the token for a request, providing it sends the response back with the same token.

is because the token is not a static field; it can be modified along the way when a packet travels across a network, for example by a proxy. In Figure 4.3 we can see how the CoAP Client sends a request with the token 'ABC', the CoAP proxy changes this token to '123' and relays the request to the intended CoAP server. The server processes the request and responds with token '123'. When the CoAP proxy receives a response with token '123' it maps the token back to 'ABC' and relays the response to the CoAP client. The client will receive a response with token 'ABC' that correctly will map to the request token of 'ABC'. If we were to integrity protect the tokens this CoAP functionality would break, therefore the token must be omitted from the authenticated data.

So, some data in a packet is not protected by cryptographic operations, some data is encrypted and integrity protected, and some data is just integrity protected. There is one more data protection concept present. This is called Additional Authenticated Data (AAD) by the OSCoAP specification. It consists of data that needs to be integrity protected, but that is not present in the sent data. This is for example the algorithm used for data protection. The algorithm does not need to be sent since both parties are aware of it from the Security Context but we still want to include it in the integrity protected data in order to assure correct behaviour and detect anomalies.

#### 4.4.6 Unprotected options

Just like some header fields can not be encrypted and integrity protected, some options can not be calculated into the tag. One example is the Proxy-Uri option which is used by a forward proxy to determine the final destination. A proxy can change the Proxy-URI and thereby invalidate the tag even though everything is working as intended. This shows the need for the possibility to leave out parts of the message from any encryption and authentication. A list of protected and unprotected options is provided in Table 4.1.

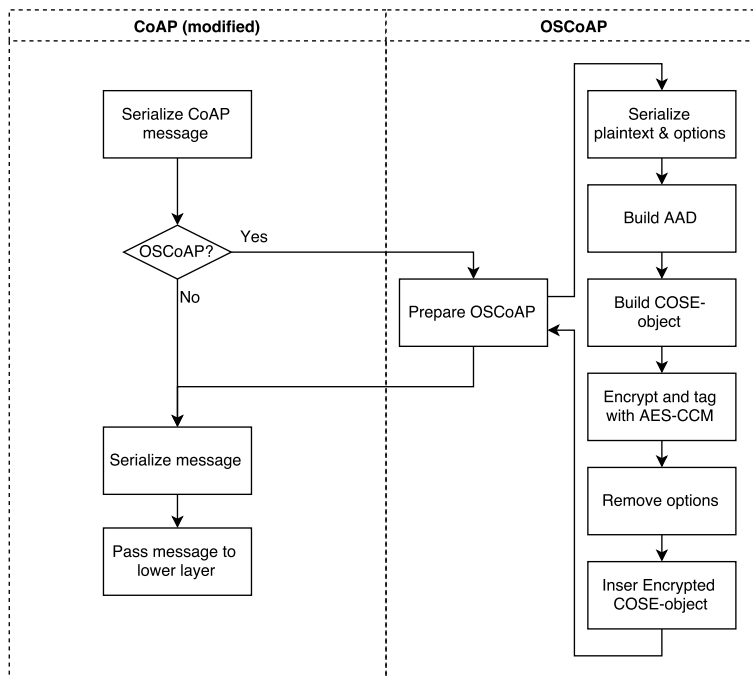
#### 4.4.7 OSCoAPs relation to other Object Security standards

There are a handful of well known object security standards. Most noteworthy are the JOSE and XML-signature standards used for securing JSON and XML respectively [40][41]. OSCoAP uses the not yet finished standard COSE, described in Section 4.2. These standards are frameworks for how to protect data but they do not say anything about what data needs what protection. This is the job for OSCoAP. These standards also do not specify how the protected data is integrated into a specific protocol like e.g. CoAP.

As mentioned before, with object security one can pick and choose what to protect. OSCoAP does precisely this for CoAP. This is a security critical task. On first thought, one might think that it is enough to protect the payload of a message. This is not the case, certain header fields and other protocol related functionality can be very important to protect. How OSCoAP protects important parts of CoAP is described in the following chapters.

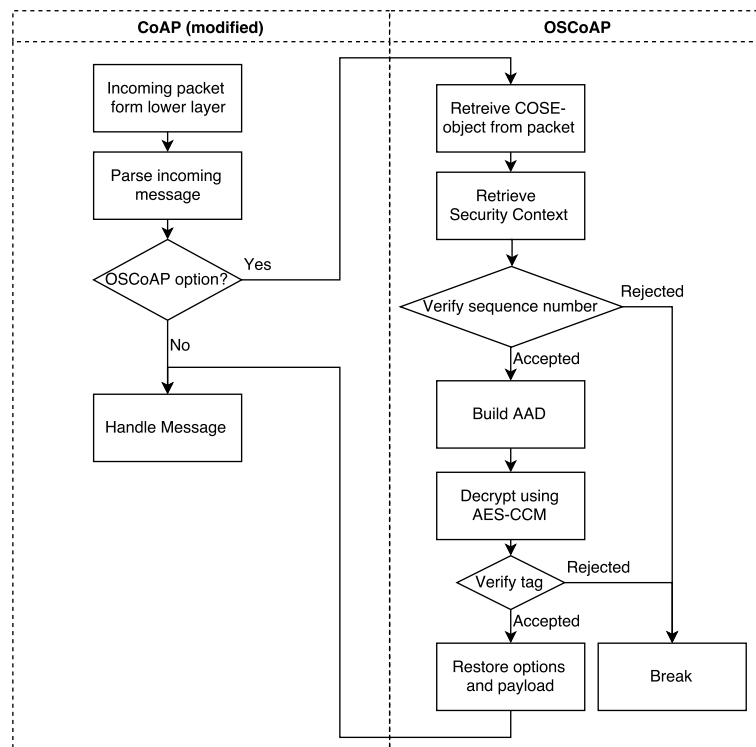
# Protocol implementation

This section will broadly describe the implementation of OSCoAP. We provide a C implementation for embedded platforms and a Java implementation for desktops or back-end servers. The implementations are both extensions of existing libraries. These libraries together with the inherent difference between Java and C for embedded devices makes the implementations differ in details, the general outline of the implementations are however similar. We will begin with describing the common structure of the implementations and then focus on the specific differences between the Java and the C implementation. Focus will be on describing the functions used to serialize and parse OSCoAP messages, since that is the key difference from a plain CoAP implementation.



**Figure 5.1:** Logical flow for serializing an OSCoAP message

The serialize function, as seen in Figure 5.1, is called when an outgoing message is sent. It takes a CoAP/OSCoAP message in an internal representation and converts it into a binary representation that can be passed to lower layers in the protocol stack and be transmitted over the network. The serialize function will look whether the message has the object security option set. If it is not set, the message is an ordinary CoAP message and will be serialized as usual. If the option is set, a function to prepare the OSCoAP message will be called. The function will remove the options as specified in Table 4.1 together with an eventual payload. It then places those options and eventual payload as the contents of a COSE-object. Then the AAD will be constructed using parameters from the Security Context. Also retrieved from the Security Context is the key and the nonce that is then used to encrypt the COSE-object and integrity protect the AAD. The encrypted COSE-object will then be placed into the OSCoAP message after the protected options are removed. The preparation of the OSCoAP message is now complete. The internal representation of the OSCoAP message is passed back to the serialize function that will serialize the remainder of the OSCoAP message. When the serialization is completed the binary message is passed down to the underlying protocol to be transmitted to the destination.



**Figure 5.2:** Logical flow for parsing an OSCoAP message

The parse function, as seen in Figure 5.2, is called when a message is received. It receives data from lower layers in the protocol stack and creates an internal rep-

resentation of the CoAP/OSCoAP message. This internal representation will then be passed on to the application that handles the CoAP/OSCoAP communication. As can be seen in Figure 5.1, the serialize function is called when a lower layer has processed the underlying protocol and placed the packet in a buffer. The packet will be parsed and the data will be put in its respective place. If no OSCoAP option is encountered the parsing is complete, the produced internal representation of the message will be passed on and the parse function is complete. When an OSCoAP option is encountered, the parser will continue to parse all non encrypted data. When the parsing is complete, functions specific to OSCoAP will be called. The first step that will be taken is to retrieve the COSE-object from the message. The Context ID is then read from the COSE-object if the message is a request. If it is a response, the correct context ID can be found using the token. The Context ID is used to fetch the correct Security Context. This Security context is first used to validate the sequence number from the incoming message. If the sequence number is invalid, the operation will abort and the message will be dropped. If the sequence number is valid, the AAD will be constructed according to what type of message that is received. This AAD will be used with the key and the nonce from the Security Context to decrypt the contents from the COSE-object. The AEAD decryption also performs a validation check on the tag. If this tag is found to be invalid the packet is dropped and the parsing will terminate. If the decryption and tag validation is a success the now decrypted contents of the COSE-object will be placed into the internal representation of the OSCoAP message. When that is complete the parsing of the message is done. The internal representation of the message can then be sent to the upper layer to be handled.

## 5.1 Californium, a Java implementation

Californium is a popular Java implementation of CoAP [42]. It is wide spread, well documented and well structured so it suited us well. We have implemented the Java client as a patch for Californium. Care has been taken to change as little as possible of the original Californium library, making for an easier patch to maintain when Californium is updated. This is achieved through extending the classes that needs changing.

Californium keeps a stack of classes that processes incoming and outgoing messages. Outgoing messages from a server or client instance enters the stack at the top and sequentially travels through the stack in order to let the stack layers make appropriate changes before finally exiting at the bottom before being fed to the UDP socket. The OSCoAP protocol is implemented as such a stack layer. Since encrypting large parts of the message will make other layers operations impossible, the object security layer is placed at the bottom of the stack to not interfere. The stack can be seen in Figure 5.3.

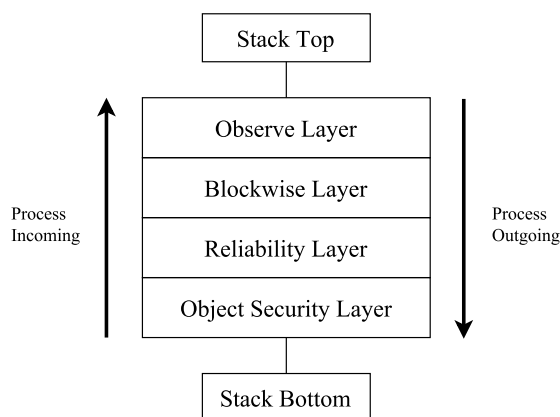
The object security layer contains 4 important methods that are triggered when a message is sent or received:

- **Outgoing request:** If an object security Option is present in the outgoing message (it will be an empty option at this stage), the message will be protected in accordance with OSCoAP. The security context that was used is



saved so that it can be correlated to the response using the token field, since the reply will come back with the same token.

- **Incoming request:** Similarly to outgoing requests, the presence of an object security option triggers the OSCoAP action. But in this case it is decryption and integrity checking. The security context is saved here too.
- **Outgoing response:** When an outgoing response occurs, the response needs to be correlated to the prior incoming request. When this is done, the correct security context can be retrieved and the response can be protected using it.
- **Incoming response:** An incoming response also needs to find the saved security context so that it can be properly decrypted and integrity checked.



**Figure 5.3:** Californium processing stack

When a client needs a context, it can get it from the URI. This can be done since a context is unique for every client and server pair and the client always initiates the communication. A server can be identified using the URI according to the CoAP RFC (“The CoAP server is identified via the generic syntax’s authority component, which includes a host component and optional UDP port number. The remainder of the URI is considered to be identifying a resource”).

When a server needs a context, the Context ID which can identify the correct context is available in the received message. The server saves this CID in the californium provided Exchange.java, which helps mapping requests to responses, so that the response can use the same CID later. The security context structure uses the naming convention of sending/receiving sequence numbers instead of server/client. This differs from OSCoAP draft 4, a discussion on why this was needed is provided in Section 8.4.

Californium uses the Maven build system, therefore all dependencies are included using Maven. Extra dependencies for OSCoAP are COSE and therefore CBOR.

## 5.2 Erbium CoAP on Contiki OS

The implementation for an embedded device was characterized by the need of a small memory footprint. Reducing the utilization of both RAM and Flash memory was of highest priority. This had large impact on design decisions we took. The embedded device we used was a fairly capable device. The Texas Instrument cc2538dk System on Chip development board has an ARM Cortex-M3 CPU clocked at 32 MHz, 32 KB of RAM and 512 KB of Flash. It also features a IEEE 802.4.15 radio module. This System on Chip was seen as representative of future embedded devices and would be a suitable target to work against.

### 5.2.1 The Contiki OS

Contiki is an operating system developed by Adam Dunkels at SICS, which was designed for connected constrained devices [4]. With a full TCP/IP stack it will use around 30 KB of Flash memory and 10 KB of RAM. Contiki is also fully open-source and free to use. This has made it a popular choice for developers of IoT devices.

The Contiki OS uses Proto-Threads, a type of light-weight threads developed by Adam Dunkels. The Proto-Threads all share a common stack and are non-preemptable. This means that cooperative multi-threading has to be used. Although simple, they provide a solution to multitasking with minimal overhead in both memory and CPU time.

The operating system is written in C and has support for a multitude of processor architectures, among them x86, AVR, MSP430 and ARM. The Cooja network simulator is a part of the Contiki ecosystem. Cooja can simulate hardware nodes and communication between these nodes. We had hoped to use Cooja to aid in our development but due to trouble in the compiler chain we were unable to use it.

### 5.2.2 The Erbium CoAP Library

The Erbium [43] CoAP library was implemented by Matthias Kovatsch at ETH Zurich. It includes a REST engine to handle REST-resources, the resources can be implemented by a developer using predefined functions and are easily connected to the engine. This makes it easy for a developer to implement an application with a CoAP interface using the Erbium Library. Having a REST-engine was a good feature for us since we could test our code in a proper environment. We had to make some small changes to the REST-engine. We modified the REST-engine to respond to OSCoAP requests with messages protected by OSCoAP.

### 5.2.3 Additional libraries used

OSCoAP uses COSE structures to format the encrypted part of the messages. Instead of writing a COSE implementation from scratch we used COSE-C [44] by Jim Schaad. That in turn uses the CN-CBOR [32] library for its CBOR structures and OpenSSL for cryptographic primitives. The CN-CBOR library is written for

constrained nodes, hence the name. It proved to be very efficient in both RAM and Flash usage.

We did not think the OpenSSL library would be the best choice for an embedded device and instead used mbedTLS. We figured that OpenSSL would not work well on an embedded platform, since it is mainly used in desktop or server applications. Instead we chose a library developed for use in constrained devices. The mbedTLS [45] library was formerly developed under the name PolarSSL until ARM took over and changed the name to mbedTLS for their mbed program [46]. We only implemented support for AES-CCM-64-64-128 in COSE, since that is required by the OSCoAP draft. The mbedTLS library is optimized for embedded devices, but the AES implementation stores large tables for the S-boxes. These tables takes up a significant amount of Flash storage, about 9 KB, but doing finite field calculations on the fly would be too processor intense for a constrained node.

The COSE library, although written in C is not optimized for use on embedded devices. Several large buffers are statically allocated and used for serialization of COSE structures. These buffers alone attributed to 40 KB of both RAM and Flash. This is clearly unacceptable as the devices we used was limited to 32 KB of RAM. We had to rewrite parts of the COSE library to use a single buffer. We reduced the size of that buffer to save further space. This resulted in 0.5 KB being used for this buffering functionality.

#### 5.2.4 The Erbium based OSCoAP Implementation

We wanted to alter as little as possible in the Erbium CoAP library to preserve functionality and minimize work. Therefore, the OSCoAP extension of er-CoAP was implemented with simplicity in mind. We identified two functions `coap_parse_message` and `coap_serialize_message` that would serve as step in points for our implementation. The overall structure of parsing a OSCoAP message can be seen in Figure 5.2.

### 5.3 Deviations from the OSCoAP draft

Our Erbium extension does not implement any client functionality. It is trivial to extend the functionality for clients as well but it is not necessary for our intended tests. We also do not fully implement duplication of the Observe-option. This is however only necessary for application using observe, which our tests do not do.

More important, in our OSCoAP implementations, we found that the function of sequence numbers to act as server or client sequence numbers imposed some restrictions on functionality. Therefore, we have changed the convention to sender and receiver sequence number, regardless of client/server role. This is discussed further in Section 8.4

We also elected not to implement the sliding window since that feature of the OSCoAP draft is not of primary interest to our tests.

---

# Quantitative methodology

---

In this chapter we discuss the methodology of the quantitative analysis. The performed tests compare OSCoAP with CoAP and CoAP over DTLS. All protocols are tested when running over UDP/IPv6. We test the C code on a physical constrained device, specifically Texas Instruments cc2538dk [47], with 802.15.4 and 6LoWPAN carrying the IPv6 packets [27]. The Java code runs on a desktop OS using standard IPv6 for packet delivery.

## 6.1 Earlier tests

Earlier work by Palombini, Object Security in the Internet of Things [11], have measured CPU-time, memory footprint and network activity for integrity protection of OSCoAP messages. Earlier tests on CoAP over DTLS have been made by Raza et al. [6].

### 6.1.1 Similarities

The C part of these tests were performed on the same hardware as Palombini tests. These tests also use the same protocols in the network stack.

### 6.1.2 Extensions and differences

The most important difference is that Palombinis results measures the performance of an earlier draft of OSCoAP [10]. In the new draft 4, OSCoAP utilizes COSE and CBOR for message structure and representation instead of defining a representation itself. Palombinis tests were also carried out using an earlier version of the Contiki operating system (version 2.7). We have made efforts to make our tests comparable to these results but with an extended scope.

Our tests measure the performance in a similar way but since our code implements OSCoAP draft 4, the messages will be encrypted as well as integrity protected. This will likely generate more overhead in memory footprint since we need additional cryptographic libraries and operations. It will not necessarily generate more overhead at the network layer, but data representation and encoding is

very different in this thesis compared to Palombinis. We make use of the COSE draft, while Palombini was forced to implement her own data representation.

We also want to compare our implementation to the current state of the art security protocol, DTLS.

## 6.2 Testing environment

### 6.2.1 Hardware and simulators

We had hoped to use the Contiki Cooja simulator which can emulate physical nodes and the communication between nodes. The simulator has support for AVR and MSP430 architectures. But we had trouble with the build tools for these architectures. We had to settle for testing network overhead with the Java version over the local loop-back interface on a PC. Wireshark is a popular program for dissecting network packets and it has support for CoAP, this suited our needs fine.

For performance tests on the cc2538dk development boards we used the real-time timer from the Contiki OS. They proved to be easy to use and gave us an acceptable precision for our measurements.

For communication between Java on a desktop and C on a cc2538dk, we used what is called a border router (BR). The BR acts as translator between IP and IP over 6LoWPAN. This way, we could route packets between a desktop computer running the Java implementation and a cc2538dk board using the C implementation.

### 6.2.2 Software dependencies

The software dependencies for these tests are:

- Contiki 3.0[48]
- CN-CBOR[32]
- COSE-C[44]
- mbedTLS[45]
- Californium[42]
- Scandium - now a part of Californium
- tinyDTLS[49]

## 6.3 Methodology

### 6.3.1 Scope and scenario

OSCoAP focuses on scenario 3.1 and 3.2 listed in [50]. The first scenario, scenario 3.1, is a critical request possibly routed through a proxy, for example the query “is the door locked?”. Here you get a single response and the freshness of that

response is of the utmost importance. The client needs to be sure the response is the correct one and that the request and response has not been tampered with.

The second scenario, scenario 3.2, is a publish-subscribe scenario where the client sends a single request and obtains multiple responses periodically. This might be security critical for a temperature gauge at a factory.

For simplicity, our tests will focus on the one-request-one-response scenario, i.e. scenario 3.1. The data for this scenario is the core tests needed to validate the suitability of OSCoAP compared to other approaches.

It is also sufficient to limit the test to include only GET requests since the only significant difference to other actions is the amount of payload in the request/response. The responses of our requests will carry a payload and therefore that aspect will be covered anyway.

OSCoAP is superior to DTLS in the aspect of not limiting the use of a proxy to the same extent as DTLS will. However, a proxy would not affect the tests in a relevant way so one will not be used during the tests.

### 6.3.2 Tested protocols

The protocols compared in these tests are:

- CoAP
- OSCoAP
- CoAP over DTLS

Tests will be carried out to measure:

- Network overhead.
- Network latency.
- CPU-time for serializing and parsing OSCoAP.
- Memory footprint.

### 6.3.3 Limitations

In order to make the tests simple, well defined and reproducible, we have introduced the following limitations:

- DTLS will be measured without any performance enhancing add-ons, such as session resumption
- The communicating parties have pre shared keys both in DTLS and OSCoAP.
- Resource discovery is assumed to already have taken place. This enables hard coding of addresses in the tests.
- Resource authorization is implicitly granted.
- No CoAP proxies exists in the network.

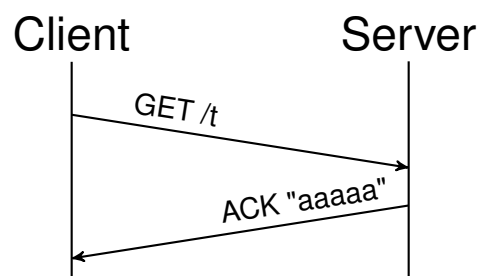


In this chapter we present the tests and the obtained results. All tests compare OSCoAP to plain CoAP over UDP and CoAP over DTLS. We test and measure network overhead, CPU-time and memory footprint. Every test is first described and then the results are presented.

## 7.1 Network overhead

### 7.1.1 Tests performed

The network overhead tests are quite simple in nature. Since we are interested in data from network protocols running on top of UDP, we can simply send packets over the loopback interface on a desktop computer and use Wireshark to measure the network traffic. We can use the non constrained standard stack for these tests since only overhead is of interest, depicted in Figure 3.2. The network traffic is depicted in Figure 7.1. It consists of a GET request from the client and the corresponding response from the server including a payload.



**Figure 7.1:** One request, one response scenario

The setup is very similar for all three protocols. The Uri-Path option is present in all of the protocols, even if encrypted. OSCoAP additionally includes the Object Security option and the Max-Age option. The token length is fixed to 2 bytes. In these tests, the sequence number used by OSCoAP only uses values that can be represented by a single byte.



## 7.1.2 Results

### Constant overhead

The constant overhead on the non constrained stack consists of the IP and UDP overhead. All stated packet sizes are gross sizes and includes this overhead of 32 bytes.

### DTLS handshake overhead

The DTLS handshake in our DTLS-PSK setup is of constant size. The handshake size will vary depending on supported modes, algorithms etc. In our case, it consists of the messages presented in Table 7.1.

**Table 7.1:** DTLS Handshake sizes

Client Hello	125 Bytes
Hello Verify Request	92 Bytes
Client Hello	157 Bytes
Server Hello, Server Hello Done	152 Bytes
Client Key Exchange, Change Cipher Spec, Finished	127 Bytes
Change Cipher Spec, Finished	99 bytes

It can be made smaller by allowing fewer extensions and cipher suites but this gives a rough estimate. There will always be at least 6 flights in a DTLS handshake.

### Request overhead

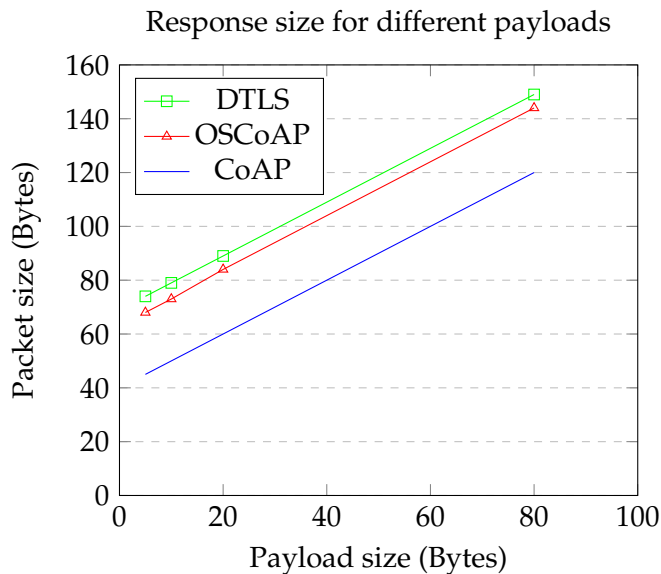
The requests carry no payload in these tests and are therefore of constant size. The request sizes are listed in Table 7.2.

**Table 7.2:** Sizes for GET requests and responses with 5 byte payload

	CoAP	OSCoAP	DTLS/CoAP
Request Size (Bytes)	40	66	69
Response Size (Bytes)	45	68	74

### Response overhead

The total packet sizes from different payload sizes in the response is plotted in Figure 7.2. The correlation between the payload size and the packet size is linear, as expected. In relation to a plain CoAP packet, an OSCoAP response packet will carry 23 bytes of overhead and a DTLS/CoAP packet will carry 29 bytes of overhead.



**Figure 7.2:** Response network overhead

The difference in overhead between OSCoAP requests and responses is due to different use of COSE. An OSCoAP request includes an extra field containing the key identifier, hence 3 extra bytes is added to the overhead.

### 7.1.3 Related tests

Lithe [6] is a DTLS integration library for CoAP which compresses DTLS headers when they are sent over 6LoWPAN [6]. Lithe reduces the number of bits sent both for the handshake and for every message in the record layer. In the record layer, which is what is comparable to OSCoAP, the reduction is 8 bytes. This makes the overhead of Lithe-DTLS 2 bytes lower than for CoAP responses instead of 6 bytes higher. For CoAP requests Lithe-DTLS will have 5 bytes lower overhead.

## 7.2 CPU time

### 7.2.1 Tests performed

Described here are the tests for the CPU-time on the cc2538dk hardware. The test were conducted with use of Contiki's real-time clock. We chose to limit our tests to the relevant critical parts of the execution. The handling of incoming and outgoing messages are where the protocols will differ. We tested the original Erbium CoAP implementation against our OSCoAP implementation and CoAP over DTLS by measuring the time needed by the following functions.

- parse The function parse takes an incoming message and decodes it into a data representation of the message. If OSCoAP is used it will handle decryption and verification of the COSE object. It will then restore the CoAP packet to the standard data representation.
- serialize The function takes a CoAP packet in data-representation form and transform it to binary that can be sent to lower layers of the communication stack. If OSCoAP is used it will create a COSE object, encrypt and tag the COSE object.

We also chose to make tests on the encryption and decryption methods to better be able to analyse the results. The Contiki real-time clock measure time in system ticks. One second on the cc2538dk platform is 32768 ticks. This gives us a good resolution to evaluate our results. We elected to run the tests 20 times and calculate an average from the tests.

We knew that the OSCoAP implementation uses heap allocated memory. And that it therefore will suffer from decreased performance. We decided that tests of the heap allocation functions and some other functions, for example memset, would yield interesting results. These functions can easily be avoided in a more optimised implementation of OSCoAP.

## 7.2.2 Results

Table 7.3 presents the results of the tests. Each test was run 20 times. The average execution time was calculated and ticks was then converted into milliseconds. Table 7.4 shows the execution time for the dynamic memory functions for OSCoAP.

**Table 7.3:** Execution time of selected functions

	Parse	Serialize	Decrypt	Encrypt
CoAP	0.0427 ms	0.0412 ms	N/A	N/A
OSCoAP	2.028 ms	2.668 ms	1.123 ms	1.327 ms
DTLS	0.836 ms	1.285 ms	0.8102 ms	0.8716 ms
DTLS + CoAP	0.879 ms	1.326 ms	0.8102 ms	0.8716 ms

**Table 7.4:** Execution time for memory functions

OSCoAP	Parse	Serialize
Execution time	0.275 ms	0.443 ms

## 7.3 Memory footprint

### 7.3.1 Tests performed

We measured the memory footprint with the objdump tool [51]. We chose to compare the Erbium-CoAP server from Contiki with modified versions that uses

OSCoAP and CoAP over DTLS. The `objdump` utility reads the section header of an ELF-file (Executable Linkable Format) a standard executable file. The elf section header contains information about the segments of the executable binary file.

- `.bss` Contains uninitialized data, if a program needs a 128 byte buffer and does not initiate it to a value it will take up 128 bytes of the `.bss` segment.
- `.data` Contains data initialized to a value. If a program needs the static variable  $a = 42$  the `.data` segment will contain this.
- `.text` Contains the actual machine code. This is the instructions that a CPU will execute.

Since the OSCoAP implementation uses heap allocated memory we had to manually calculate the run-time memory usage. We did this by reading the code and sum up all times memory was dynamically allocated. Also of interest was the code size of the libraries we included. Namely for CBOR, COSE, and mbedTLS. We were also interested about the code size of the functions for dynamic memory. We used the `nm` tool to dump the ELF file and manually summed up the code size of the functions for these libraries.

### 7.3.2 Results

Table 7.5 contains the numbers for the server using CoAP and OSCoAP. From these numbers RAM and ROM (Flash) usage on the actual device can be calculated. The RAM usage is calculated by: `.data + .bss = RAM`. ROM usage is calculated by `.data + .text = ROM`.

**Table 7.5:** Memory footprint in bytes for a server using CoAP, OSCoAP and CoAP over DTLS

Application:	Server		
Protocol:	CoAP	OSCoAP	CoAP + DTLS
<code>.bss</code>	13799	14031	14899
<code>.data</code>	1772	2788	1922
<code>.text</code>	47070	77895	74498
RAM ( <code>.data + .bss</code> )	15571	16819	16821
ROM ( <code>.data + .text</code> )	48842	80683	76420

Table 7.6 shows the minimum amount of heap allocated memory needed for the functions parse and serialize OSCoAP messages. Note that these functions will never be called at the same time.

**Table 7.6:** OSCoAP minimum heap memory allocated at runtime

	Parse	Serialize
Heap allocated Memory	315 Bytes	524 Bytes

Table 7.7 shows the breakdown of selected parts of the OSCoAP ELF-file.

**Table 7.7:** Code size for different parts of OSCoAP

	COSE-C	CN-CBOR	mbedTLS	dynamic memory
Code size:	4770 Bytes	1394 Bytes	3734 Bytes	1714 Bytes

The main objective of this thesis was to research whether draft 4 of OSCoAP was functional in a practical scenario. This involves both assessing the functionality and the performance. For the functionality, it is very clear that the current draft is functional within our tested scope, except for the sequence number issue discussed in Section 8.4. The performance is judged by a more continuous scale. We find it sufficient for most scenarios.

This chapter discusses the suitability of OSCoAP compared to plain CoAP and DTLS/CoAP in light of the findings in previous chapters. It starts by discussing the quantitative findings in the tests and then goes on to discuss qualitative aspects of OSCoAP.

## 8.1 Network overhead

### 8.1.1 Performance comparison

In our tests, OSCoAP used more overhead than plain CoAP, which was of no surprise. A somewhat more surprising result was that OSCoAP had a slightly lower network overhead than DTLS/CoAP even for the record layer. This means that OSCoAP is often preferable to standard DTLS when network overhead is an important aspect.

### 8.1.2 Lithe or OSCoAP

We have not made any tests on Lithe but Raza et al. provides comparable data in [6]. In terms of network overhead, Lithe seems to be the strongest competition for OSCoAP since Lithe has a lower overhead in the record layer. This means that there is a threshold for when Lithe is suitable compared to OSCoAP. The overhead for Lithe is the sum of the handshake messages and the overhead produced by the record layer. For OSCoAP the overhead consists solely of the message overhead. This gives the equation

$$x * o < HS + x * r$$

for when to use OSCoAP, where  $HS$  is the total size of the compressed handshake,  $r$  is the record layer overhead,  $o$  is the OSCoAP message overhead and  $x$  is the

expected number of packets sent for each handshake. Note that the number of packets is the important measure, not the amount of data.

Lithe is however not suitable in some applications where it would be more efficient than OSCoAP. The main problem with Lithe is that it uses an extension of 6LoWPAN to compress the DTLS headers. This has consequences for devices communicating with devices outside the 6LoWPAN network. In order to do this using Lithe, the 6LoWPAN border router needs to implement the Lithe 6LoWPAN extension. This might not be the case for a mobile device using different border routers. OSCoAP however, is network agnostic; the intermediate nodes need not be aware of the existence of OSCoAP.

### 8.1.3 Network Failure rate

Another factor to consider when choosing between object security and channel security is how network drop rate impacts the success rate and in extension, the network overhead. A packet that is corrupted or dropped while traversing the network will need to be retransmitted. For OSCoAP an DTLS record layer, this is straightforward; the packet needs to be retransmitted.

For an ongoing DTLS handshake though, a lost or corrupted message can lead to several retransmits depending on how far the handshake has progressed, as described in Section 2.3.3. This makes a Connection Based security model unsuitable for situations where new connections occur frequently.

## 8.2 CPU-Time

Having longer execution time than CoAP could be expected. That encryption and decryption will take up a lot of the additional execution time was also to be expected. We expected the CPU time for serialize and parse of DTLS to be similar to OSCoAP but we found out that DTLS performed significantly better than OSCoAP. This can in part be explained by the more efficient cryptographic library that tinyDTLS uses. The encryption and decryption times for DTLS was quite a bit lower than the times for OSCoAP. Also the usage of dynamic memory in OSCoAP decreases the performance versus DTLS. DTLS does not use dynamic memory, instead it uses a block allocator for a few things like session data. This is clearly a better design decision, using dynamic memory clearly decreases the performance of our OSCoAP implementation.

The execution time can also be improved by utilizing the crypto co-processor.

## 8.3 Memory footprint

We used the tests of the Contiki node running a Erbium-COAP server as baseline to compare the DTLS and OSCoAP implementations. Both libraries use much more Flash memory for code than the CoAP server. This is as expected because of the increased functionality and logic needed to process more complex protocols. Both DTLS and OSCoAP use look-up-tables stored in flash memory for the AES

crypto. Together with the cryptography libraries this increases the usage of flash memory. The OSCoAP implementation uses around 4.2 KB more flash memory than DTLS implementation. The usage of generic libraries for COSE and CBOR can attribute to some of this. The implementation of OSCoAP is a prototype implementation. Quality, both in terms of code-quality and performance is lacking. Regarding the usage of memory two key points can be established.

### 8.3.1 Dynamic Memory Usage

Using functions from the C standard-library for dynamic memory on embedded systems is a bad idea in several ways. Firstly estimations of run-time memory usage are hard to make. If the heap grows out of its expected boundaries for some reason erroneous behaviour can occur. Secondly the code needed to manage the dynamic memory is rather large. In fact the code is larger than the amount of memory we dynamically allocate. This is clearly a waste of memory and can quite easily be improved in a future implementation of OSCoAP.

### 8.3.2 Unsuitable libraries

The libraries chosen for the implementations are general implementations of the COSE and CBOR formats. These libraries contains code that is not used at all in the OSCoAP implementation. This increases the ROM and RAM usage. The implementation also duplicates a great deal of code from Erbium CoAP. This is also a point for further improvement.

## 8.4 Deviations from OSCoAP draft 4

As mentioned in Section 5.3 we ran in to a problematic situation in the implementation of draft 4 of OSCoAP. The OSCoAP draft makes a clear separation between the client and the server role. The security context therefore had the fields "server sequence number" and "client sequence number". Requests would always originate from a client and responses would originate from a server. In this model, the sequence numbers will be easy to manage and consistent with the logic in the draft.

The problem arose when a single node would act as both a server and a client for the same security context. This situation can arise in a system where 2 nodes are both providing each other with information. How should a node handle the sequence numbers in a scenario like this? The behaviour was not specified by the draft and the consequences were that the client sequence number was increased both when sending and receiving requests; the same goes for the server sequence numbers.

Our solution to this was to remove the explicit role of a node as either a client or a server. Instead we said that a node is a node, the sequence numbers should really reflect whether the node sending or receiving data. Hence the new naming, sender and receiver, in the security context. This way of handling sequence numbers



is agnostic to the current role of a node, which is very practical when the role changes over time.

## 8.5 Similar approaches

As mentioned in Section 1.2, there are numerous adaptations of channel security for constrained devices. However, to the best of our knowledge, other security solutions providing the same level of functionality for using object security in constrained nodes are few and far between. The most similar solution is OSCAR [9]. OSCAR provides object security for the Internet of Things, but with focus on access control. More important, the object security format in OSCAR is designed for protection of *publish-subscribe* communication rather than *end-to-end* communication. OSCAR has a model for *many-requests-one-response*, while the model in OSCoAP is adapted for *one-request-one-response* or *one-request-many-responses*.

The OSCAR paper itself states that “[...] we need to adapt existing or future standards specifying the object security format such as CMS (Cryptographic Message Syntax) and JOSE (JSON Object Signing and Encryption) to constrained devices.”. This is precisely what OSCoAP does. OSCAR should thus not be seen as a competing protocol but rather as a complement.

## 8.6 Future Work

### 8.6.1 OSCoAP Shortcomings

#### Perfect Forward Secrecy

OSCoAP lacks a security feature that can be obtained in most channel security protocols, namely Perfect Forward Secrecy (PFS)[52]. PFS is the property of a security system that if a key is compromised, no previous communication recorded by a malicious party prior to the compromise should be affected by the compromise. This means that, you can not decrypt previous communication using the long term key. This can be important for many applications and more so when there is physical access to the node in possession of a cryptographic key.

#### Multicast

There is currently no functionality supporting multicast of encrypted messages in OSCoAP. All communication is modelled on the one sender, one receiver, scenario.

#### Application Layer Fragmentation

The latest OSCoAP draft, version 4, does not discuss the support for blockwise options, a CoAP extension for packet fragmentation on the application layer [53].

## 8.6.2 Suggested future research

### Perfect Forward Secrecy

Work is on the way to implement this functionality for OSCoAP [54]. This work uses a handshake to set up ephemeral keys used to obtain PFS, which would remove some of the advantages OSCoAP has compared to protocols that depends on handshakes. As a complement to PFS obtained through handshakes, we suggest looking into the feasibility of obtaining PFS by using pre-keys [55] or puncturable encryption [56]. These solutions require no handshake.

### Multicast

Multicast needs to be further researched if implemented. A multicast scenario puts new requirements for sequence number handling and forward/backward secrecy since it is effectively a type of group communication.

### Application Layer Fragmentation

For more confidence of the viability of OSCoAP we suggest further tests of application layer fragmentation. Testing the Blockwise option, that allows fragmentation of CoAP packets at the application layer, would give interesting results to compare with fragmentation on lower layers.

### Further Extensions

There is a paper by Castellani et al. that explores using EXI over CoAP for constrained devices [57]. With XML having a prominent position as a web data format alongside JSON, it might be worth looking into complementing COSE with XML signatures over EXI.

## 8.6.3 Suggested optimizations

Regarding the implementation of OSCoAP for Contiki there are a few improvements that can be done to increase the performance significantly. The first improvement is to get rid of any usage of dynamic memory. Contiki provides a block allocator that can be used for the few cases where dynamic memory is needed. The second improvement is on the same subject. Both the cn-cbor library and in extension COSE-C uses dynamic memory. By implementing a limited subset of COSE and CBOR without dynamic memory, both ROM and RAM can be saved. The execution time can also be expected to improve. The last improvement would be to evaluate and incorporate a more efficient crypto library. Judging by the crypto library used by tinyDTLS this can probably save both ROM and execution time.



# Conclusion

---

We have argued that the current situation for constrained nodes calls for a partial transition to object security due to asynchronous communication. We then set out to explore the feasibility of such a system, namely OSCoAP. The most important goal was to obtain proxying properties for intermediate nodes without hindering end-to-end security. Performance was not the end goal, though it is an important measure for feasibility in constrained nodes.

An implementation of OSCoAP was presented and tested; the tests show that object security is not only feasible in terms of functionality, but often preferable to channel security in terms of network overhead. With the solution leaving room for improvements in memory footprint and CPU-time, the viability of OSCoAP can be further increased.

We also suggest a few extensions and changes to the OSCoAP protocol for increased functionality, among them a more flexible role definition for clients and servers.



---

## References

---

- [1] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)." RFC 7252 (Proposed Standard), June 2014.
- [2] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2." RFC 6347 (Proposed Standard), Jan. 2012. Updated by RFC 7507.
- [3] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, "Object security of coap (oscoap)," Internet-Draft draft-selander-ace-object-security-04, IETF Secretariat, March 2016. <http://www.ietf.org/internet-drafts/draft-selander-ace-object-security-04.txt>.
- [4] "Contiki Operating System homepage." <https://www.contiki-os.org/>.
- [5] S. Raza, S. Duquennoy, J. HÅglund, 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*, vol. 7, no. 12, pp. 2654–2668, 2014.
- [6] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt, "Lithe: Lightweight secure coap for the internet of things," *IEEE Sensors Journal*, vol. 13, pp. 3711–3720, Oct 2013.
- [7] R. Hummen, J. H. Ziegeldorf, H. Shafagh, S. Raza, and K. Wehrle, "Towards viable certificate-based authentication for the internet of things," in *Proceedings of the 2Nd ACM Workshop on Hot Topics on Wireless Network Security and Privacy, HotWiSec '13*, (New York, NY, USA), pp. 37–42, ACM, 2013.
- [8] M. Sethi, J. Arkko, and A. KerÅnen, "End-to-end security for sleepy smart object networks," in *Local Computer Networks Workshops (LCN Workshops), 2012 IEEE 37th Conference on*, pp. 964–972, Oct 2012.
- [9] M. Vucinic, B. Tourancheau, F. Rousseau, A. Duda, L. Damon, and R. Guizzetti, "OSCAR: Object Security Architecture for the Internet of Things," in *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, (Sydney, Australia), June 2014. <https://hal.inria.fr/hal-00985976>.

- [10] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, "Object security of coap (oscoap)," Internet-Draft draft-selander-ace-object-security-02, IETF Secretariat, June 2015. <http://www.ietf.org/internet-drafts/draft-selander-ace-object-security-02.txt>.
- [11] F. Palombini, "Object security in the internet of things," Master's thesis, KTH, School of Information and Communication Technology (ICT), 2015.
- [12] E. Rescorla and B. Korver, "Guidelines for Writing RFC Text on Security Considerations." RFC 3552 (Best Current Practice), July 2003.
- [13] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks." RFC 7228 (Informational), May 2014.
- [14] J. Vasseur, "Terms Used in Routing for Low-Power and Lossy Networks." RFC 7102 (Informational), Jan. 2014.
- [15] C. B. Margi, M. A. S. Jr., M. Naslund, B. T. de Oliveira, P. S. L. M. Barreto, R. Gold, G. T. de Sousa, and T. C. M. B. Carvalho, "Impact of operating systems on wireless sensor networks (security) applications and testbeds," in *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pp. 1–6, Aug 2010.
- [16] L. Schmertmann, K. Hartke, and C. Bormann, "Cotdls: Dtls handshakes over coap," Internet-Draft draft-schmertmann-dice-cotdls-01, IETF Secretariat, August 2014. <http://www.ietf.org/internet-drafts/draft-schmertmann-dice-cotdls-01.txt>.
- [17] P. Eronen and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)." RFC 4279 (Proposed Standard), Dec. 2005.
- [18] O. Garcia-Morchon, S. L. Keoh, S. Kumar, P. Moreno-Sanchez, F. Vidal-Meca, and J. H. Ziegeldorf, "Securing the ip-based internet of things with hip and dtls," in *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '13*, (New York, NY, USA), pp. 119–124, ACM, 2013.
- [19] R. B. et al., "Secure communication for smart iot objects: Protocol stacks, use cases and practical examples," tech. rep., IEEE, 2012.
- [20] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [21] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing." RFC 7230 (Proposed Standard), June 2014.
- [22] 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, 7465, 7507, 7568, 7627, 7685.
- [23] J. Postel, "User Datagram Protocol." RFC 768 (INTERNET STANDARD), Aug. 1980.

- [24] J. Postel, "Transmission Control Protocol." RFC 793 (INTERNET STANDARD), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.
- [25] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460 (Draft Standard), Dec. 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112.
- [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 RFCs 6282, 6775.
- [27] J. A. Gutierrez, E. H. Callaway, and R. Barrett, *IEEE 802.15.4 Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensor Networks*. New York, NY, USA: IEEE Standards Office, 2003.
- [28] "Ace working group ietf." <https://datatracker.ietf.org/wg/ace/documents>.
- [29] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)." RFC 7049 (Proposed Standard), Oct. 2013.
- [30] T. Bray, F. Yergeau, E. Maler, J. Paoli, and M. Sperberg-McQueen, "Extensible markup language (XML) 1.0 (fifth edition)," W3C recommendation, W3C, Nov. 2008. <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- [31] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format." RFC 7159 (Proposed Standard), Mar. 2014.
- [32] "cn-cbor, constrained node cbor implementation." <https://github.com/cabo/cn-cbor>.
- [33] "cmp, messagepack library witten in c." <https://github.com/camgunz/cmp>.
- [34] "Bson, binary json specification version 1.0." <http://bsonspec.org/spec.html>.
- [35] "Messagepack." <https://github.com/msgpack/msgpack/blob/master/spec.md>.
- [36] R. Kyusakov, J. Schneider, T. Kamiya, and D. Peintner, "Efficient XML interchange (EXI) format 1.0 (second edition)," W3C recommendation, W3C, Feb. 2014. <http://www.w3.org/TR/2014/REC-exi-20140211/>.
- [37] J. Schaad, "Cbor encoded message syntax," Internet-Draft draft-ietf-cose-msg-12, IETF Secretariat, May 2016. <http://www.ietf.org/internet-drafts/draft-ietf-cose-msg-12.txt>.
- [38] D. Whiting, R. Housley, and N. Ferguson, "Counter with CBC-MAC (CCM)." RFC 3610 (Informational), Sept. 2003.
- [39] D. McGrew, "An Interface and Algorithms for Authenticated Encryption." RFC 5116 (Proposed Standard), Jan. 2008.
- [40] M. Miller, "Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE)." RFC 7520 (Informational), May 2015.



- [41] J. Reagle, T. Roessler, F. Hirsch, D. Solo, and D. Eastlake, "XML signature syntax and processing (second edition)," W3C recommendation, W3C, June 2008. <http://www.w3.org/TR/2008/REC-xmlsig-core-20080610/>.
- [42] "Californium, coap in java." <http://www.eclipse.org/californium/>.
- [43] J. Schaad, "Erbium (er) rest engine and coap implementation for contiki." <http://people.inf.ethz.ch/mkovatsc/erbium.php>.
- [44] "Cose-c, cose implementation in c, using cn-cbor." <https://github.com/cose-wg/COSE-C>.
- [45] "mbedtls, arm." <https://tls.mbed.org/>.
- [46] "Arm mbed iot platform." <https://www.mbed.com/en/>.
- [47] "Texas instruments cc2538dk." [www.ti.com/tool/cc2538dk](http://www.ti.com/tool/cc2538dk).
- [48] "Contiki os official source code." <https://github.com/contiki-os/contiki>.
- [49] "tinydtls, eclipse project for dtls in c." <https://projects.eclipse.org/projects/iot.tinydtls>.
- [50] G. Selander, F. Palombini, K. Hartke, and L. Seitz, "Requirements for coap end-to-end security," Internet-Draft draft-hartke-core-e2e-security-reqs-00, IETF Secretariat, March 2016. <http://www.ietf.org/internet-drafts/draft-hartke-core-e2e-security-reqs-00.txt>.
- [51] *objdump(1)*, *Linux manual page*.
- [52] H. Krawczyk, *Perfect Forward Secrecy*, pp. 457–458. Boston, MA: Springer US, 2005.
- [53] C. Bormann and Z. Shelby, "Block-wise transfers in coap," Internet-Draft draft-ietf-core-block-20, IETF Secretariat, April 2016. <http://www.ietf.org/internet-drafts/draft-ietf-core-block-20.txt>.
- [54] G. Selander, J. Mattsson, and F. Palombini, "Ephemeral diffie-hellman over cose (edhoc)," Internet-Draft draft-selander-ace-cose-ecdhe-01, IETF Secretariat, April 2016. <http://www.ietf.org/internet-drafts/draft-selander-ace-cose-ecdhe-01.txt>.
- [55] M. Marlinspike, "Forward secrecy for asynchronous message." <https://whispersystems.org/blog/asynchronous-security/>. Accessed: 2016-06-01.
- [56] M. D. Green and I. Miers, "Forward secure asynchronous messaging from puncturable encryption," in *2015 IEEE Symposium on Security and Privacy*, pp. 305–320, May 2015.
- [57] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi, "Web services for the internet of things through coap and exi," in *2011 IEEE International Conference on Communications Workshops (ICC)*, pp. 1–6, June 2011.



**LUND**  
UNIVERSITY

Series of Master's theses  
Department of Electrical and Information Technology  
LU/LTH-EIT 2016-532

<http://www.eit.lth.se>